English exam

Solution sketches for Written exam ID1020 Algorithms and Data structures

Re-exam 2019-12-18

Examiner: Robert Rönngren
Allowed aiding material: None

Each sheet of paper handed in should have the following information:
1. Name and social security number (personnummer)
2. Number for the question
3. Page number

Only write solutions to a single question on each sheet of paper (does not apply for the multiple choice questions special answer sheet).

The exam consists of two parts. Part I covers five areas:
- The fundamentals
- Sorting
- Searching
- Graphs
- Applications

with two questions per area for the grade E.

Part II, which is not compulsory, contains four questions for grade A-D.

Solutions to each question will be graded with 0-10 points.

To pass Part I, i.e. get grade (E), you need a minimum of 12 points per area in Part I. For Fx the requirement is a minimum of 12 points on four of the areas and a minimum of 10 points on the remaining area. You need to pass Part I to be able to obtain a higher grade (A-D). Points from Part I is not transferred to Part II.

OBS! Points, answers and grading:

> **For multiple choice questions**: Identify correct and incorrect propositions. Each correctly identified proposition gives 1 point. 10 points are awarded if all eight propositions are correctly. Your answers should be marked and handed in on the special answer sheet.
>
> **For the other questions:** ***Complete solutions with motivations/explanations are required. That is, answers without motivations are normally graded with 0 points.***

If you have reached grade E on Part I the following apply for higher grades from the score on Part II.

_ Grade D: 5-10 points
_ Grade C: 11-20 points
_ Grade B: 21-30 points
_ Grade A: 31-40 points

Solution suggestions will be posted on Canvas.

Part 1

The Fundamentals:

Multiple choice question: The Fundamentals question 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. An algorithm for which big-Oh and big-Omega are defined also has big-Theta defined
   FALSE, Theta is defined iff big-Oh and big-Omega are within a constant of each other

2. Amortized time complexity for an algorithm/data structure is the same thing as the average of the big-Oh time complexities for the different operations of the API of the algorithm/data structure
   FALSE, big-Oh defines the worst case, while amortized is an average over a number of operations. Cp. resizeing-arrays where big-Oh would be defined by the case where an operation forces a resize resulting in big-Oh(N). While the amortized access time can be proven to be constant as the linear operation will only occur at most at every Nth operation while the others will be constant time.

3. LIFO and FILO queueing policies are equivalent
   TRUE

4. An array based list implementation of a queue cannot be used to implement circular queues/circular buffers
   FALSE

5. The big-Omega time complexity for a linked-list based priority-queue is big-Omega(1) as insertions, deletions and searches on a queue with a single element always can be performed in constant time
   FALSE, big-Oh, big-Omega and big-Theta are asymptotic bounds, that is bounds that they come close to when the size of the problem (N) grows to be large. Thus big.Omega is not defined by problem sizes that are small (or of size one)

6. To determine the largest problem size that is possible to solve on a specific computer system with a specific algorithm, it is sufficient to know the big-Oh time complexity of the algorithm
   FALSE, the size of a problem that can be solved on a specific computer system is not only defined by the execution time but also the memory requirements

7. A recursive algorithm which divides a problem so that the same problem (parameters to the recursive call) occurs more than once in the recursion will never terminate
   FALSE, a counter example is Fibonacci numbers where a naïve implementation will re-calculate the same numbers repeatedly but it will not execute forever (indefinitely)

8. Recursion is often used to implement divide-and-conquer algorithms
   TRUE

The Fundamentals question 2:

State the time complexity for big-Oh, big-Omega and big-Theta and memory complexity (extra memory used by the algorithm) for the code sections below. All variables, constants and arrays have integer data types if nothing else is states and `K` and `L` are large positive integer numbers.

Many students miss points on this section (and others) since they does not provide any explanation/motivation for how they have calculated the complexities. Another thing many miss on is that you do not express the complexity with the specific variable/constant names, i.e. K*L is N^2 as the names of the variables are not important for the performance. Also many students just seems to look for if there is a for- or while-loop without checking the input, i.e. assuming a loop always will execute N-times while the number of times the loop depends on the input. Checking the input will not only yield correct answers but also greatly facilitates the analysis in some of the cases.

a) 2p

```
i=j=0;
while(i++ < K)
  while(j++ < L)
    print(a[i]*b[j]);
```

The complexity depends on both the combination of statements but also of the input. In this case the outer while-loop will run K-1 times. However the inner while-loop will only execute once as the loop-variable **`j`** is not reset after the first execution of the loop. So the total number of times the print statement will be executed is L-1. Thus the complexity is: big-Oh is N, big-Omega is N, big-Theta is N, memory O(1)
However, had **`j`** been reset to zero in each iteration of the outer while-loop the complexity would have been: big-Oh is N^2, big-Omega is N^2, big-Theta is N^2, memory O(1)

b) 2p

```
for(i=0; i<K; i++)
  print(sum(a,i));
…
int sum(int a[K][L],k)
{ int i=-1, s=0;
  while(++i < L) s += a[k][i];
  return s;
}
```

The for-loop will execute K times and in each iteration sum() will be called. sum() executes a while loop L times in each call. Thus big-Oh is N^2, big-Omega is N^2, big-Theta is N^2, memory O(1)

c) 2p

```
for(i=0;i<K;i++)
  for(j=i; j<K; j*=2)
      a[i] = b[i][j];
```

The inner loop is infinite as `j == i ==` 0. Multiplying `j` with 2 in each iteration still gives zero. Hence neither big-Oh, big-Omega or big-Theta is defined. Memory usage is constant (O(1))

If **`i`** (and implicitely **`j`**) would have been initialized to a positive value the complexity would have been big-Oh is Nlog(N), big-Omega is Nlog(N), big-Theta is Nlog(N) as the inner for-loop where `j` is multiplied by two in each iteration only would be executed log(K) (log(N)) times, memory O(1)

d) 2p

```
int summa = strangeSum(K);

int strangeSum(int k)
{
   if(k<=2) return k;
   return strangeSum(k-1);
}
```

`strangeSum(int k)` will return `k`. However it will perform k recursive calls. Thus, big-Oh is N, big-Omega is N, big-Theta is N, and memory usage is O(N) unless the language/compiler can perform tail-recursion optimization and reuse the activation record on the stack so that memory usage becomes O(1)

e) 2p

```
for(i=0; i<K; i*=2)
{ int j = 0;
  while(j++ < K)
    for(k=j; j < L;)
      return i*j*k+a[j];
}
```

The return statement will force the execution in the method/function to terminate when it is executed which will happen in the first iteration. Hence, big-Oh is 1, big-Omega is 1, big-Theta is 1, memory O(1) **.** This is independent of the fact that the outer loop is infinite as **`i`** is initialized to zero (cp. question c).

Had **`i`** been initialized to a positive number and if we had not had a return statement as the inner-most statement the analysis would have been a little bit different: Then the outer loop would have been O(log(N)) and the inner loops would have both been O(N) resulting in a total time-complexity of O(N^2log(N))

Sorting

Multiple choice question: Sorting 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. The number of operations used by Insertionsort to sort the elements of an array is on the same order as the number of inversions in the array
   FALSE: if there are no inversions it would still do N operations

2. A sorting algorithm cannot be both in-place and stable
   FALSE

3. At most one inversion can be removed from an array by switching places (swapping) of two elements in the array
   FALSE: By moving an element passed several other elements one can remove more than one inversion in one swap. This is why there are more efficient algorithms than for instance Insertion-sort which only swaps adjacent elements. Merge- ad quick-sort move elements longer distances. However, swapping over longer distances also opens the possibility that one reorder elements with equal keys.

4. An in-place sorting algorithm maintains the relative order of elements with equal keys
   FALSE

5. Any stable sorting method has a big-Theta performance bound
   FALSE

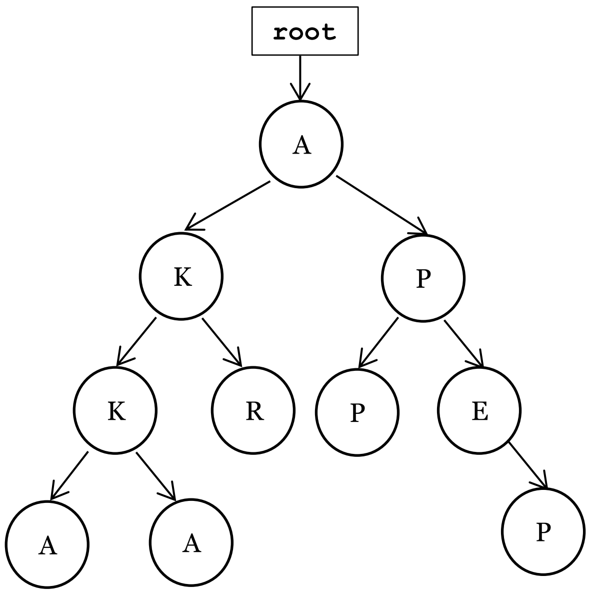
6. Given knowledge of the properties of the input it is sometimes possible to create sorting methods which do not rely on comparisons
   TRUE: for instance one can sort a number of integer values in a known range [K,L] in linear time by just checking the frequencies of each number in the range

7. A sorting method based on a balanced search tree would have a time complexity of O(Nlog(N))
   TRUE

8. A binary tree is heap-ordered if the keys in each node's both children (if any) are less than or equal to the key in that node (the parent node)
   TRUE

Sorting question 2:

Given the tree built from linked nodes below:



a) Implement data structures in JAVA that can be used to implement the tree above. The data structure need only be a single class and a method to instantiate objects from the class. 2p

```
…. class Element
{
  private char c;
  Element left, right;

  public Element(char tkn)
  {
    this.c = tkn; this.left = this.right = null;
  }
}
```

b) Implement a method which takes a reference to the root of a tree and prints the contents of the tree. The order in which the content should be printed should be such that if the root of the tree above (`root`) is the parameter, the method should print the content of the tree in the order PEPPARKAKA (gingerbread) 2p

```
… void printTree(Element root)
{
 if(root != null)
 {  printTree(root.right);
    StdOut.printf("%c",root.c);
    printTree(root.left);
}
```

c) State the big-Omega, big-Oh and big-Theta time and memory complexity for the method to print the contents of the tree? 2p

big-Oh = big-Omega = big-Theta = N, time complexity is N as the method goes through each of the N elements in the tree to print them. Memory complexity depends on the depth of the recursion, i.e. the depth of the tree which in the best case is balanced which results in big-Omega(log(N)) and in the worst case has grown as a linked list giving a worst case of big-Oh(N).

Given the code segment below:

```
void sort(int a[], int length)              //length == length of the array a[]
{
  int i, j;
  for(i=1; i<length; i++)                   // for-loop 1
    for(j = i; j > 0 && a[j] < a[j-1]; j--)  // for-loop 2
    { int t = a[j];
      a[j] = a[j-1];
      a[j-1] = t;
    }
}
```

d) Define a loop-invariant for the algorithm above and show how it can be used to prove the correctness of the algorithm. 2p

The algorithm is Insertion-sort.

Loop-invariant*: All elements in the array on indices lower than i are sorted in ascending order*

The initial state is that there is only one element in the array at an index lower than i and a single element is per definition sorted. After each iteration in the loop one more element will be sorted compared to when that iteration begun. The for-loop will go through all indices in the array and finish when i has reached a value which is max index in the array +1. Hence all elements in the array will be on indices lower than i when the loop terminates and consequently all elements will be sorted in ascending order when the loop terminates.

e) Is the method above stable and/or in-place? Motivate the answer! 2p

The algorithm does not use more than three extra integer variables – hence it is in-place.

The algorithm only swaps adjacent elements and does not swap elements with equal values – hence it is stable.

Searching

Muliple choice question: Searching 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. A good and effective way to map a 32-bit hashcode to a 12-bit hash value, is to use the 12 most significant bits (i.e. the leftmost 12 bits) of the 32-bit hashcode as the hash value
   FALSE: This will not result in uniform hashing

2. A hash table based on linear probing has the following big-Oh access times: insert O(1), delete(N), search (lookup) O(1)
   FALSE: Worst case performance is O(N) while the expected performance (with reasonably uniform hashing) is O(1)

3. In a hash table based on separate chaining the array should not be less than 1/8 full to ensure that searches (lookups) on average can be performed in constant time
   FALSE: A less full-array would still give constant time accesses (assuming uniform hashing) but it would waste memory

4. Binary search in an ordered array will find a specific element in O(log(N)) time if the element is present in the array and can determine that an element is not present in O(N) time
   FALSE: Finding an element and determining that an element is not present are both O(log(N)) operations

5. The time complexity of an ordinary binary search tree (not red-black or 2-3-tree) is O(N) for insertions and O(log(N)) for look-ups
   FALSE

6. A symbol table that supports ordered operations are thread-safe, meaning that it supports that parallel operations could be performed on the ST without errors
   FALSE: No a standard symbol table is not thread safe

7. To find all <key,value> pairs in a range (i.e. the ***i***th to the ***i***+***n***th elements) in a ST is an O(*n*) operation in a binary search hash table where *n* is the number of elements in the range and the size of the ST is N
   TRUE

8. When using modular hashing of positive integer hashcodes, the array length should be selected as a prime number if possible
   `TRUE`

Searching question 2:

a) Assume you have an application in which you are using a symbol table. Also assume the number of unique keys exceed 500. Explain under what circumstances you should select to use a symbol table based on: i) a binary search tree (BST), ii) a left-leaning red-black binary search tree or iii) a hash table 3p
   i) For most applications you should avoid BSTs for this large amount of keys as it has O(N) time complexity. Only if you need simpler code and know that the tree will be reasonably balanced could it be used. (for smaller data sets it could possibly be used if ordered operations are needed)
   ii) Preferable to BSTs but use it only if: ordered operations are required as it has O(log(N)) time complexity; or if one cannot calculate efficient hash codes; or if the application cannot tolerate individual operations which takes longer time as the case is for operations on a hash table which requires a resize operation. (Hash tables normally are more efficient in terms of amortized/average access times and use similar amounts of memory)
   iii) If ordered operations are not required and the application can tolerate occasional operations that are O(N) (i.e. operations involving resizing) as it can be expected to have O(1) average time complexity for most operations

b) Define, in JAVA or C, a function/method for binary search in an array with unique integer keys sorted in ascending order. The length of the array is `length`. 2p
   See the book page 8

c) What are the big-Oh, big-Omega and big-Theta (if it exists) time and memory complexity for your code from question b). Explain how you have calculated them. 2p

Big-Omega is 1 if the key searched for is found at the middle index of the array, big-Oh is log(N) as the size of the part of the array searched in, is halved in each iteration, big-Theta is not defined. The algorithm use only two extra integer variables, hence the memory complexity is constant (1)

d) Explain the concept of ”uniform” hashing and why it may be an important property. 1p

For the definition see the book page 463 Assumption J. Uniform hashing means that the hash values are uniformly distributed on the elements of the array in the ST. That is, it is equally probable that a value hashes to any of the positions in the array. It is the underlying assumption for hash-tables having O(1) insert, look-up and delete average time complexities as this gives a probabilistic guarantee that there will be a low number of collisions.

e) Define, in JAVA, C or pseudo code, methods that can be used to assess how uniform hashing of elements of the type DATA is. Assume that the hash maps to the interval [0,N] 2p

See the figure on top of page 463 in the book. Sketch: create an integer array of size N. .Then calculate the hash values for a typical set of unique keys that are substantially larger than N (preferably at least 100*N) and increment the value at the index in the integer array that the key hashes to. If all possible hash values have similar frequencies and are evenly spread then the hash is near uniform (for an exact analysis you need the course on statistics…). (think about why the keys should be unique). Alternatively one could use a hash table and check the number of collisions, for instance the lengths of the lists in a ST based on separate chaining (in which case we do not need to use unique keys) where the lengths of the lists should be near similar for all lists.

Graphs

Multiple choice question: Graphs 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. A strongly connected component of an undirected graph has edges directly connecting all pairs of vertices in the component
   FALSE: You need not have direct edges, the connections can be indirect (through a chain of other vertices)

2. The shortest path between any two vertices in an edge-weighted directed graph is always unique
   FALSE: You can have parallel paths which are of equal length

3. The concept of a shortest paths tree is only defined for edge weighted graphs
   FALSE, defined also for unweighted undirected graphs which can be seen as an edge weighted graph with unit edge weights

4. A bipartite graph is a graph consisting of two strongly connected components where the two components are connected by only one edge in each direction
   FALSE

5. There may be more than one shortest path tree with the root in a specific vertex in an edge weighted graph with arbitrary edge weights
   TRUE

6. A data structure similar to that used to implement a hash table based on linear probing is an efficient structure for implementing a sparse graph
   FALSE, this would be less efficient than the data structure similar to a hash table with separate chaining

7. Dijkstra's algorithm is the same as depth-first search in case we have a graph with unit edge weights
   FALSE

8. The critical path is defined as the longest path in a shortest-path tree
   FALSE

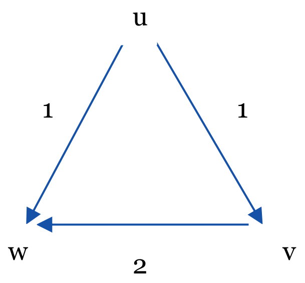
Graphs question 2:

a) Assume we have modelled a communication network as an edge weighted graph. The edge weights represents the cost to send a message over the link represented by the edge. Which algorithm(s) of the algorithms covered in the course are suitable to apply to calculate how to send a message from a vertex to all other reachable vertices at the lowest cost? 2p
   To minimize the cost to reach all possible vertices we should calculate a minimum spanning tree which is done by Prim's and Kruskal's algorithms

b) Show how the critical path for a scheduling problem, modelled as an edge weighted directed acyclic graph, can be calculated. State the big-Oh time complexity for the algorithm. 3p
   See the book pages 664-666

c) Show if the following statement is true or false:
"In a directed graph the shortest path between any two vertices, ***v*** *and* ***w****, must be part of any Shortest Path Tree with the root in vertex* ***u****"* 2p

FALSE, Proof by a simple counterexample with three vertices connected in a triangular shape by three edges. The shortest path tree with root in u consists of the edges u->w and u->v with a total weight of 2. A tree with edges u->v and v->w would have weight 3 and the path from u->w would not be the shortest path from u to w. Hence, the shortest path from v->w is not part of the shortest path tree.



Some (many) students tend to complicate the counter-examples, sometimes to an extent where the intended proof becomes to complex or obfuscated to be valid.

d) Show what graph properties must hold for it to be possible to calculate a topological order. Show why these properties are necessary (you need not show why they are sufficient) 3p

The graph must be acyclic and directed. A topological sort is a sort of graph thus we must be able to compare two vertices along a path to see which comes first, i.e. to we must be able to define a total order between vertices. We need not be able to order vertices on parallel paths. If we have bidirectional edges, i.e. un un-directed graph we cannot determine an order between vertices. In a directed graph the starting vertex of an edge comes before the ending vertex. If we have cycles we cannot order the vertices. As an example assume we have vertices w and v where there is an edge w -> v and an edge w -> v, in this case we cannot determine which comes first of v and w. These requirements also ensure that there will be at least on vertex with no incoming edges and one with no outgoing edges (i.e. one possible starting and one possible end-point for the topological order) Hence these are necessary properties.

Applications

Multiple choice question: Applications 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. The `Comparable` interface in JAVA is an example of a language design to support, among other things, sorting methods based on call-back
   TRUE

2. By using memoization, a recursive solution to compute Fibonacci numbers can compute the Nth Fibonacci number in <= O(N) time
   TRUE

3. A Heap structure is always balanced
   TRUE

4. A FIFO-queue is a type of priority queue where the priorities are implicit
   TRUE (implicit: capable of being understood from something else though unexpressed) The order of the elements in a FIFO-queue depends on when they were inserted to the queue. This time need not be explicit in the elements, but the order implicitly depends on this (non-visible) priority.

5. Sorting algorithms with a big-Oh time complexity of O(N^2) should never be used to sort large amounts of data
   FALSE: Example: Insertion sort is efficient when sorting data with few inversions as the big-Omega complexity is O(N) although the big-Oh complexity is O(N^2)

6. Symbol Tables are often used as a key element to implement efficient large scale sparse matrix operations
   TRUE

7. Symbol Tables can reduce the execution time of many of today's large scale problems by magnitudes in the order of billions
   TRUE (see the last paragraph in the book on Searching, Applications. Or simply compare the execution time for a brute force search in a set of some billion keys to that of a look-up in a balanced search tree or a hash table, the former is O(N) while the latter are O(log(N)) and O(1) respectively)

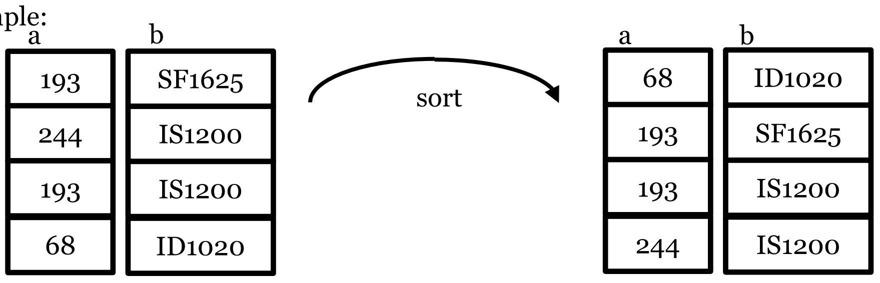
8. Quicksort with CUTOFF to Insertionsort is both stable and in-place
   FALSE: Quicksort is not stable

Applications question 2:

a) Given data in two separate one-dimensional arrays representing data of the type <studentId, course code>. The first array contains integer student-Ids and the second contains course-code (text) for a course the student is registered on.

   In JAVA, devise a method to sort the data in ascending order by student-Id. The method should be stable and in-place, which you should show. The big-Oh time complexity should be less or equal to O(N^2). 4p

   Example:



This is a variant of insertion sort. Simply copy the code from the Sorting e) assignment and co-sort the course array together with the student id array. Obs! That you do need to modify the algorithm in this way for it to work (i.e. for you to get points for the silution). Proof of properties are the same as for Sorting e) with the exception that this algorithm uses one more extra variable/array which does not affect the proof.

```
void sort(int a[], char *b[], int length)              //length == length of the
array a[]
{
  int i, j;
  for(i=1; i<length; i++)                          // for-loop 1
    for(j = i; j > 0 && a[j] < a[j-1]; j--)  // for-loop 2
    { int t = a[j];
      char *s = b[j];
      a[j] = a[j-1];
      a[j-1] = t;
      b[j] = b[j-1];
      b[j-1] = s;
    }
}
```

b,c,d) –Assume you have a large data set where each element has a unique integer key. Describe how you can do the following in the most effective way (i.e. describe the algorithms) for each of the data structures below (b,c,d):
  i) add a new element with a unique key
  ii) look-up (find) an element with a specific key
  iii) remove an element with a specific key
You should also state the time complexity for the worst, best and amortized cases for these operations.

b) Data is stored unordered in a "resizing"-array 2p

Assumption: All elements are kept at the beginning of the array. We keep track of the number of elements in the array (n) which means that we can directly identify the position where to insert a new element. Let N be the length of the array. If n < N then there is a free space at index n in the array.

Insert:
Best case: to add an element to an array with an empty index: constant time O(1)
Worst case: to add an element to an array with no free space triggers a resize: linear time O(N)
Amortized time to add an element is O(1) – proof see resizing arrays in the book

Look-up:
Best case: Linear search and the key is on index = 0: constant time O(1)
Worst case: Linear search from the beginning where key is on the last index: linear time O(N)
Amortized (in this case average): Search half of the elements: linear time O(N)

Remove is implemented by a look-up. If the key (i.e. the element) is found we remove it from the array and move the last element to that place and update n. Hence the time complexity is the same as for the look-up cases plus a linear time in the worst case if we have to resize (shrink) the array. However, a linear time plus a linear time in the worst case is still a linear time operation Best case: O(1), worst case: O(N), Amortized O(N).

(note: using hashing and linear probing introduces an order of the elements in the array which no longer makes the array elements unordered)

c) Data is stored ordered by the key in a linked list priority queue 2p

insert, look-up and remove all start with a look-up of the place where the element is to be inserted or where it is. This is implemented as a linear search from the beginning of the list. In the best case the element/place to insert in, is found at the beginning of the list in which case the insert/look-up/remove is done in constant time. The worst case is when the place/element is last in the list in which case we have a linear search time. The average case is when we have to search half of the array which is still linear time. An insertion and removal also consist of setting a limited, small number of references/pointers which is O(1). Thus for insert, look-up and remove operations we have: Best case:O(1), worst and amortized case: O(N)

d) Data is ordered by the key in a left-leaning red-black binary search tree 2p

For a full description see the book. The time complexity for these operations are all logarithmic with the exception of the best case for a look-up. In the case the element that we search for is found in the root of the tree, then the look-up time is constant, i.e. big-Omega is O(1).

Part 2

Complete solutions with motivations are required for all assignments. When grading, the time and memory complexity of your solutions will be important.

Question 2.1:

Given is a social network with N members with unique names where each member is friend with everyone in the network (directly or indirectly). Associated with the network is a log-file with timestamps. The log shows when two members became friends. The log-file is ordered in ascending order by timestamps and contains M (direct) friend connections. Friendship is an equivalence relation. You can assume that N and M are known.

Design an algorithm which calculates the earliest time when all members in the network became friends, directly or indirectly, with all other members. Also state the time and memory complexity for the algorithm. 10p

Algorithm sketch: The problem can be solved by going through the log file and add a connection at a time and check whether all members are part of the same connected component. The timestamp of the last connection added which makes all members part of the same component is the time when all members in the network became friends. This can be solved by for instance weighted quick-union with path compression

Question 2.2:

Describe how a spell-correcting system could be implemented. Given is a CSV-file with pairs of `<miss-spelled word, correct spelled word>`. Each misspelling only occurs once in the CSV-file. Your algorithm should read the text to correct from `stdin` and print the corrected text to `stdout`. Be careful when selecting algorithms and data structures. Also state the time and memory complexity for the algorithm. 10p

Algorithm sketch: Read the CSV file into a hash based ST. Use the miss-spelled words as keys and the correct spelled words as values. Then read the input and look-up every word read in the ST. If the word read is found in the ST then replace it in the output by the value (correct spelled word)

Question 2.3:

Given the following input: [76, 54, 32, 19, 16, 7, 44, 61]

a) Show, step-by-step, how a binary search tree is built given the input above 2p
See the book

b) Show, step-by-step, how a left-leaning red-black binary search tree is built given the input above 8p
See the book

Question 2.4:

Design a program (algorithm) to match sellers and buyers on a market. Assume there is only one comodity traded at the market. The program should take as input data on the format `<sell/buy, seller id/buyer id, max/min price>` where `sell/buy` is coded as [0,1], `seller id/buyer id` is a unique integer and `max/min price` is a positive integer value. Your program should have methods to continuously accept input on the format above. It should print matching seller/buyer pairs with agreed prices as soon as any such become available.

You should clearly define the criteria you use to find matching sellers, buyers and prices. 10p

A solution to this problem strongly depends on how the market works. There are several options for how to match seller offers and buyer bids.

The simplest solution is to store seller offers and buyer bids in two separate priority queues where high buyer bids have high priority while low priced seller offers have high priority and where the order between same priced bids and offers respectively are FIFO in the respective buyer bids/seller offers queues.

When a new seller offer is entered to the system one checks to see if there is a buyer bid which is higher. If there is a bid which is the lowest bid higher than the offer, that bid wins and is removed from the bid priority queue and matched with the offer. Otherwise the offer is introduced in the offer priority queue.

When a new bid is entered to the system one checks the offer priority queue and selects the first offer which has a price lower than the bid, if such an offer exists, and removes the offer from the priority queue and matches it with the bid. If no matching offer exists, the bid is entered in the bid priority queue.

This is likely to be an algorithm that could be used for most market places. It strives to maximize the number of transactions that can be completed. However, there are alternative algorithms that for example could attempt to maximize the profit for sellers by always matching the highest bid to the lowest offer available. Another thing to possibly take into consideration is for how long time a bid/offer is valid (i.e. if it has an expiration time when it should be removed from the system if it has not been part of a match of seller offer and buyer bid)