Solution suggestions[1] to

Written exam in ID1020 Algorithms and Data structures

Monday 2018-10-22 8.00-12.00

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)

There are two parts in the exam. Part I consists of ten questions for the grade E. Part II, which is not compulsory, consists of four questions for grade A-D. Each question can give up to ten points

To pass Part I, i.e. get grade (E), you need 60 points on Part I. You need to pass Part I to be able to obtain a higher grade (A-D). Points from Part I are not transferred to Part II.

If you have reached at least 60 points 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

Solutions will be posted on Canvas

[1] Suggestions – meaning that there may be different solutions to problems and that we cannot cover (all) solutions in detail

Part 1 A

Identify true and false statements. Each correctly identified statement (true or false) gives +1point, if all 8 statements are correctly identified the score on the question is 10 points.

The answers to the questions in this section of Part 1 should be marked and handed in on the answer sheet found after Part 1 A.

Question 1: Fundamentals

1. Recursion always uses stack space on the order of the depth of the recursion

   FALSE: this is not *always* true. Counter example: Many compilers use constant memory for tail-recursive methods

2. The possibility to declare methods, variables `private/public` are important to enable the creation of well-defined APIs (Application Programming Interfaces)

   TRUE: An API should hide details that the user need not/should not be able to see/access/use, hence the possibility to define what is private/public – not visisble/visible is important

3. Given that the same elements have been inserted into a LIFO queue and a stack, then a `dequeue`() operation on the queue and a `pop()` operation on the stack would return the same element

   TRUE: A LIFO queue and a stack is the same

4. A linked list implementation of a queue always has less memory overhead than an array implementation

   FALSE: A linked list would use at least one and possibly two or more additional pointers/references per item in the list while for an array the memory overhead depends on how full the array is

5. To make an Abstract Data Type (ADT) `Iterable` one must implement a means to create an object used to iterate over the elements of the ADT (i.e. an object which returns the next element for each iteration until there are no more elements in the ADT)

   TRUE: In JAVA (Iterable is a keyword of JAVA) one creates a small object to hold the state information of the iteration, for example which element in a list one has reached in this iteration

6. An algorithm where every Nth operation is O(N) operation and the remaining operations are O(1) has an amortized time complexity of O(1)

   TRUE: Amortizing one O(N) operation over (N-1) O(1) operations has an amortized time complexity of O(1), (N/(N-1) is close to 1 as N grows towards infinity)

7. A specific algorithm can be proven to be both Big-Theta(N) and $O(N^2)$ for different inputs

FALSE: Big-Theta (N) means that the worst and best case performance both are O(N) for all inputs.

8. An optimal general algorithm to solve an NP-complete problem must have time complexity <= O(Nlog(N))

   FALSE: We cannot find general solutions to NP-complete problems

Question 2: Sorting

1. A loop-invariant can be used to verify the correctness of an algorithm by constructing a proof similar in structure to a proof by induction

   TRUE: See for example the lecture notes on Insertion and selection sort, where the invariant is that elements in the array to the left of the current index is in sorted order and when we have iterated through the whole array all elements will be to the left of the index and thus the array will be sorted

2. Selection sort is in general a better sorting method for short arrays than Insertion sort

   FALSE: Insertion sort is in general a little bit faster than Selection sort and it is also stable

3. The number of inversions in an array indicates if the elements are nearly sorted or not

   TRUE: An inversion are two elements which are not in correct order – if all elements are sorted the number of inversions is zero

4. A stable sorting method does not use more than O(log(N)) extra memory to sort the elements

   FALSE: Stability has nothing to do with memory complexity, it has to do with if the relative order of elements with equal keys are preserved or not by the sorting algorithm

5. An "in-place" sorting algorithm is a sorting algorithm for which the expected execution time is insensitive to properties of the input

   FALSE: "in-place" has to do with how much extra memory space the algorithm uses – it says nothing of the execution time

6. Mergesort is faster than quicksort due to better cache performance

   FALSE: The opposite holds – Quicksort has slightly worse theoretical performance bounds but uses the cache systems better

7. Insertion sort is slower than Mergesort and Quicksort for any input

   FALSE: Insertion sort is faster for small inputs (that's why we use CUTOFF for merge- and quicksorts) and it is near O(N) for inputs with few inversions

8. A binary heap can be used to implement a priority queue with O(log(N)) time complexity for both `enqueue` and `dequeue` operations

   TRUE: See the proposition Q on page 319

Question 3: Searching

1. A Symbol Table (ST) is used to associate keys with values

   TRUE: This is the definition of what a ST should allow us to do

2. To select a range of keys present in a ST is an example of an operation which is efficiently implemented on a hash table ST

   FALSE: A hashtable is not an ordered ST, the idea behind hashing is to uniformly distribute the hash values in an un-ordered fashion – i.e. to find all keys present in the hash table between keys x and y requires us to search in an un-ordered set of data – i.e. O(N) while an ordered ST based on a tree can do this more efficiently

3. If the number of insertions/deletions of keys are very low compared to the lookups, a ST can be efficiently implemented by a sorted array and binary search

   TRUE: Each time we insert a new key we would have to sort the array to be able to use binary search for the lookups. However, if we have few insertions the cost for sorting can be amortized over many lookups – resulting in good average/amortized performance

4. The performance of binary search trees is sensitive to the order in which <key,value> pairs are added

   TRUE: In the worst case the binary tree could degenerate into a linear list depending on the input

5. A red-black tree is defined by that the nodes on even levels (0,2,4,...) are colored black and that nodes on odd levels (1,3,5,7...) are colored red.

   FALSE: A red-black tree has operations to assert that the tree will be balanced. Simply coloring every other level red or black does not guarantee that the tree will be balanced.

6. In a left leaning red-black tree the time complexity for lookup operations is ~2lg(N) where N is the number of nodes

   TRUE: In the worst case we will have one red link per black link from the path from the root to a leaf – hence the tilde notation is correct

7. In the general case the hash values need to be near uniformly distributed for hash tables to have near O(1) time complexity for insertion of keys and lookups

   TRUE: See page 463. Uniform distribution is needed not to get too many collisions. Many collisions severely degrades the performance of a hash table.

8. Hash tables are always the best choice for implementing STs when the number of <key,value> pairs is large

   FALSE: This depends on whether or not we want to be able to do ordered operations or not (as selecting a range of keys). If we do not need to do ordered operations and we have a uniform hash – then hash tables are preferable otherwise tree based STs are better choices.

Question 4: Graphs

1. A spanning tree in a connected component of an undirected graph is a tree of paths to all vertices in the component from some source

   TRUE: This is the definition

2. Depth-first search in an undirected graph traverses edges in the order of distance to the source, i.e. first all edges of distance 0, then all edges of distance 1, then distance 2 etc. to find a path to the target node

   FALSE: The above is a description of Breadth First Search

3. A topological sort of an undirected graph can be used to find a schedule that meets precedence constraints

   FALSE: A topological sort is an order in which each edge points from one vertex earlier in the order to a vertex later in the order. Thus a topological sort only applies to directed graphs without directed cycles.

4. All pairs of vertices, `v` and `w`, in a strongly connected component in a digraph should be mutually reachable, i.e. `w` should be reachable from `v` and `v` should be reachable from `w`

   TRUE: This is the definition, see page 584

5. A minimum spanning tree in an edge weighted graph is a spanning tree where the sum of the weights of the edges is no larger than the sum of weights for any other spanning tree.

   TRUE: This is the definition, see page 604

6. If we have identified a number of nodes and edges being part of a Minimum Spanning Tree (MST) during an algorithm to find the MST, then we know that the lowest weight edge connecting a node in the tree to a node not yet being identified as being in the MST must be an edge part of the MST

   TRUE: This is the definition of the Cut-property, see page 606

7. Dijkstra's algorithm will find any shortest path from a source vertex to a sink vertex in any edge weighted digraph if such a path exists

   FALSE: Dijkstra's algorithm does not work in the case of negative edge weights

8. There are no shortest paths between any two vertices in an edge weighted digraph with at least one negative cycle

   FALSE: A shortest path may not include any vertex which is part of a negative cycle per definition (see proposition W page 669). However, this does not exclude the possibility that there may be a shortest path between some two vertices where the path does not pass through a vertex part of a negative cycle.

Question 5: Applications

1. Software caching means that we store frequently reused data calculated by an algorithm instead of recalculating the data

   TRUE: This is an example of a trade-off between memory and time usage. For instance one would typically store expensive to calculate hashcodes in an object, see page 462

2. Hashing is effective when the hash maps key values from a large, sparsely populated range of possible keys to a (much) smaller range of uniformly distributed hash values

   TRUE: This is the idea behind hashing

3. Finding negative cycles is an important application of graphs in many banking systems

   TRUE: In an exchange rate system or a tax system negative cycles typically mean that one can earn money from just moving money around in the system. For examples, see the arbitrage example on pages 679-681 and the very recent example (scandal) of how major banks have helped investors earn money from collecting tax returns in the stock market on taxes never paid in the first place.

4. It is sufficient to know only the worst-case complexity of algorithms to select the best algorithm for a given problem

   FALSE: If one knows properties of the input one can select more efficient algorithms based on their best case performance, such as selecting Insertion sort for inputs with few inversions.

5. The term sparse means that the actual data set (input) is relatively small compared to what the maximum sized data set could be

   TRUE: This is the definition of sparse. We have seen multiple examples of this in the course where the possibility to solve problems in reasonable time is based on the input being sparse, such as for STs and graph problems.

6. Quicksort is, in practice, the fastest stable sorting algorithm

   FALSE: Quicksort is not stable

7. Greedy algorithms always find globally optimal solutions

   FALSE: A greedy algorithm always makes locally optimal decisions. However, this does not guarantee that the global solution built from the local optimizations will be optimal on the global scale

8. The fact that the value associated with a key is replaced if we insert a <key,value> pair where the key already is present in the Symbol table (ST) for the basic ST algorithms makes it impossible to use the Symbol-Table Abstract Datatypes of the book to build efficient indexing applications

   FALSE: We have seen that it is possible to build index applications using this API both in examples in the book and by an example in the lab.

Answer sheet 1 A *Mark with an X if the statement is true or false*

Name:_____

Social security number/Personnummer: _____

Question 1:

Statement	1	2	3	4	5	6	7	8
True								
False								

Question 2:

Statement	1	2	3	4	5	6	7	8
True								
False								

Question 3:

Statement	1	2	3	4	5	6	7	8
True								
False								

Question 4:

Statement	1	2	3	4	5	6	7	8
True								
False								

Question 5:

Statement	1	2	3	4	5	6	7	8
True								
False								

Part 1 B

Complete solutions where every part has a motivation is required.

Question 6:

Identify the time and memory complexity (extra memory used by the algorithm) for the best, worst and amortized cases for the code segments found below using Big-Oh notation. Assume `N` is a large integer, `a[]` is a one-dimensional integer array with `N` elements and `b[][]` is a two-dimensional array of dimension `NxN`.

a) 2p

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

Worst, best and amortized time complexity is $O(N^2)$ as we loop through two nested loops each using N/2 iterations ($\sim N^2/4$). No extra memory is used.

b) 2p

```
int summa = sum(N,a);

int sum(int N, int a[])
{
   if(N<=1) return a[0];
   return a[N-1] + sum(N-1, a);
}
```

This example recursively sums the elements of the array a[] from high indices to low. The if(N<=1) statement is there to ensure that the recursion ends before the parameter N to the sum-function becomes negative.

Worst, best and amortized time complexity is O(N) as we perform N sums. The method is tail-recursive which means that the memory complexity can be O(1) if the system can optimize for this, otherwise it is O(N) which is the depth of the recursion, i.e. the number of activation records (stack frames) used for the recursion without tail-recursion optimization.

Note: Some students believe that the time complexity for the best case for the sum() method is O(1) in the case N is small. This is not true, the complexity is still O(N) but N is small (eg. 1). Compare this to if we had written the code as:

```
for(i=0; i<N;i++) sum += a[i];
```

Which essentially does exactly the same thing as sum().

Other errors include that one believes that recursive functions are O(log(N)) just because they are recursive. Time complexity only has to do with the amount of work an algorithm has to perform for an input of some size (N)

c) 2p

```
int summa = sum(N,a);

int sum(int N, int a[])
{
```

```
   if(N<=1) return a[0];
   return sum(N-1, a) + a[N-1];
}
```

As for the above, with the exception that this program is not tail-recursive. Hence it has O(N) worst, best and amortized time complexity and O(N) memory complexity.

d) 4p

```
for(i=0;i<N; i++)
  for(j=0; j<N; j++)
    crossSum(i,j,N,b);

void crossSum(int i, int j, int N, int b[N][N])
{
  int k = i, l = j;
  for(;i<N;i++) b[k][l] += b[i][l];
  for(;j<N;j++) b[k][l] += b[k][l];
}
```

Again the execution time does not depend on the input (no conditionals). Two nested for-loops with O(N) iterations each and one function called in each iteration with O(N) operations. Thus the worst, best and amortized time complexity is $O(N^3)$. It uses extra memory in the form of one activation record for the calls to `crossSum()`, O(1) memory complexity.

Question 7:

Study the code segment below.

a) 2p In which order are the elements in the list printed when `printList()` is called from `main()` compared to how the numbers were entered to `main()`? Illustrate by an example!

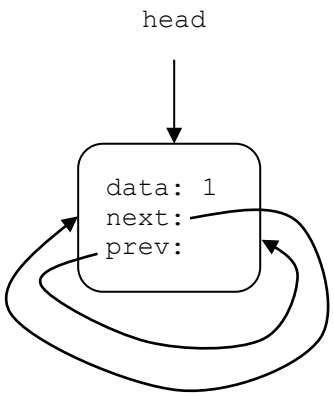
Detailed answer: The order the elements are printed in depend only on how they have been inserted into the list and in which order the list is traversed. Thus one only need to study `insertElem()` and `printList()`.

While this code is written in C (IS1200), the syntactic difference to JAVA for the parts defining the ADT, i.e. the data structure and algorithms is minimal. That is the abstract data type and methods remain the same regardless of the programming language. In JAVA these methods and the data type for an element could have been written as:

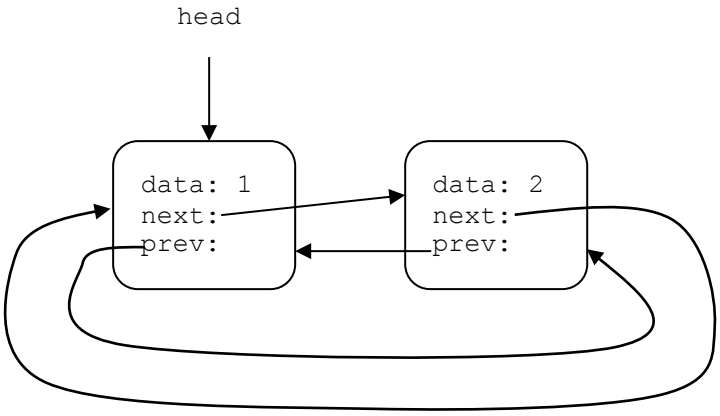
```
private class elemPtr {        // In Java one probably would have
int data;                      // named the class Element rather than elemPtr
elemPtr next, prev;
}

public void insertElem(elemPtr point, elemPtr newPtr){
  newPtr.prev = point;
  newPtr.next = point.next;
  point.next.prev = newPtr;
  point.next = newPtr;
}

public void printList(elemPtr head) {  // DO NOT MODIFY THIS CODE
  elemPtr p = head.next;
  printElem(head);
  while(p != head) {
   printElem(p);
   p = p.next;
}}
```

The ADT is a double linked circular list, which starts at the element referred to by the variable `head` in `main()`. Since the question is about in what order the elements are printed one should start checking `printList()`. `printList()` prints the element pointed to `head` and then traverses the list by following the `next` pointers/references. Thus the elements are printed from the first element to the last as expected.

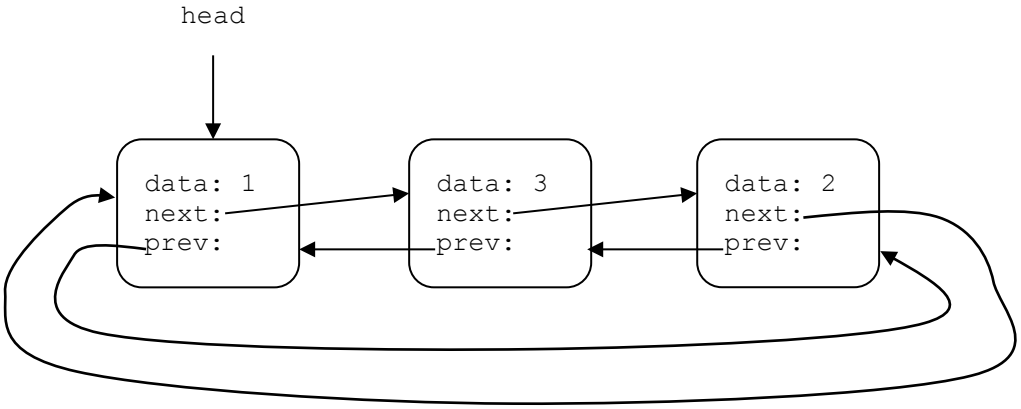
Next we check how the elements are inserted by the calls to `insertElem()`. Assume we insert 1, 2, 3, 0. Inserting 1 will create a list with one element pointed to by `head`.



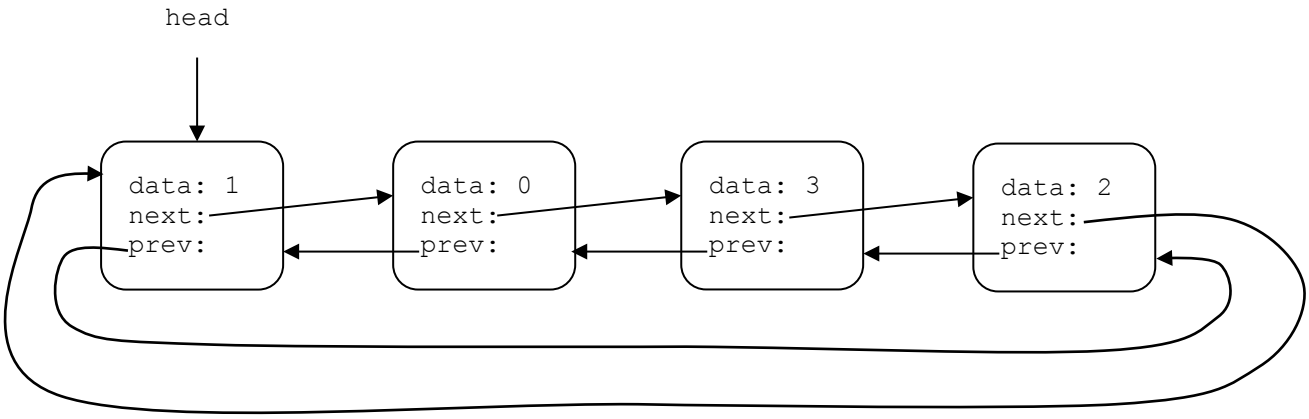
The second element, 2, will be inserted as the element following `head` (i.e. as the element pointed to `head->next`)



The third element, 3, will again be inserted directly after `head` (i.e. as the element pointed to `head->next`)



Finally an element containing 0 will be inserted



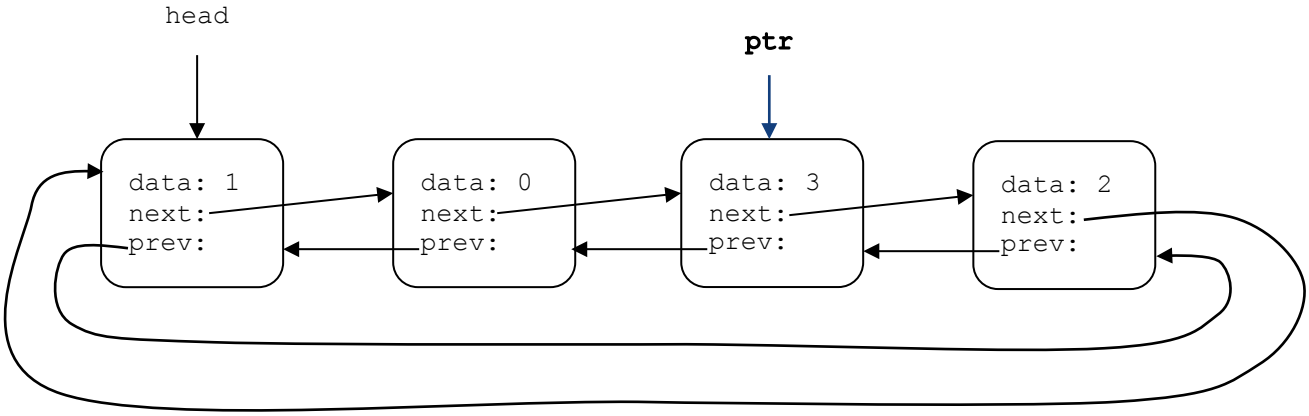
Thus, when the elements are printed starting at the element pointed to by `head` and then following the `next` pointers/references the elements will be printed in the order: 1, 0, 3, 2

b) 2p Show how the code, except `main()`, for the list must be modified if it is to be used as an implemention of a "bag". (you need not add the possibility to iterate)

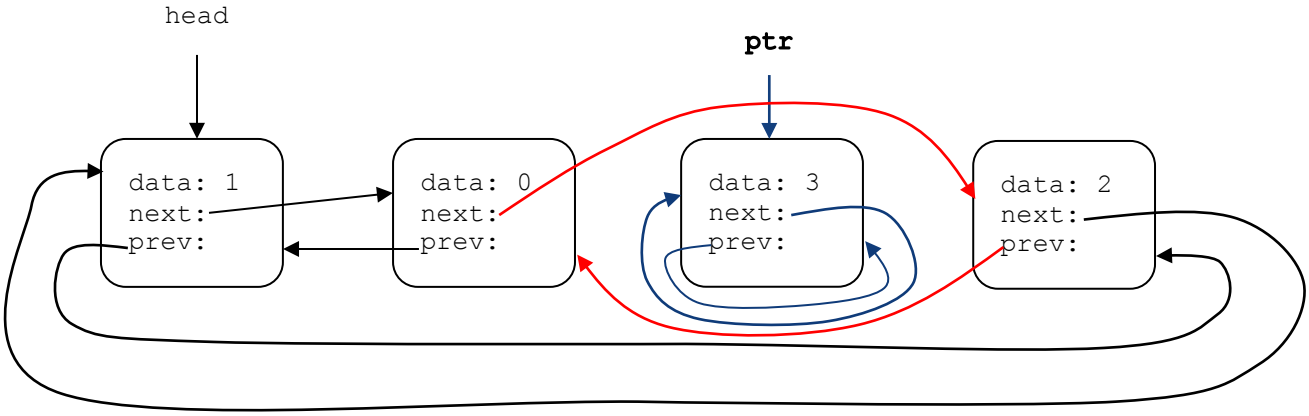
Answer: Since a bag only is a collection of items/elements without any specified order, one need not change anything in the code

c) 3p Complete the function `removeElem()` to remove and return a pointer/reference to an element in the list pointed to by the pointer/reference `ptr`. Assume that `ptr` does not point to the `head` of the list.

Answer: Assume we want to remove the element pointed to by `ptr`



We do this in two steps: first we unlink the element from the list and then we reset the `next` and `prev` pointers of the unlinked element to either point to the element itself (to make the element a small circular list itself) or point to NULL (to avoid leaving pointers into the list from the removed element). Finally we return the pointer to the element.

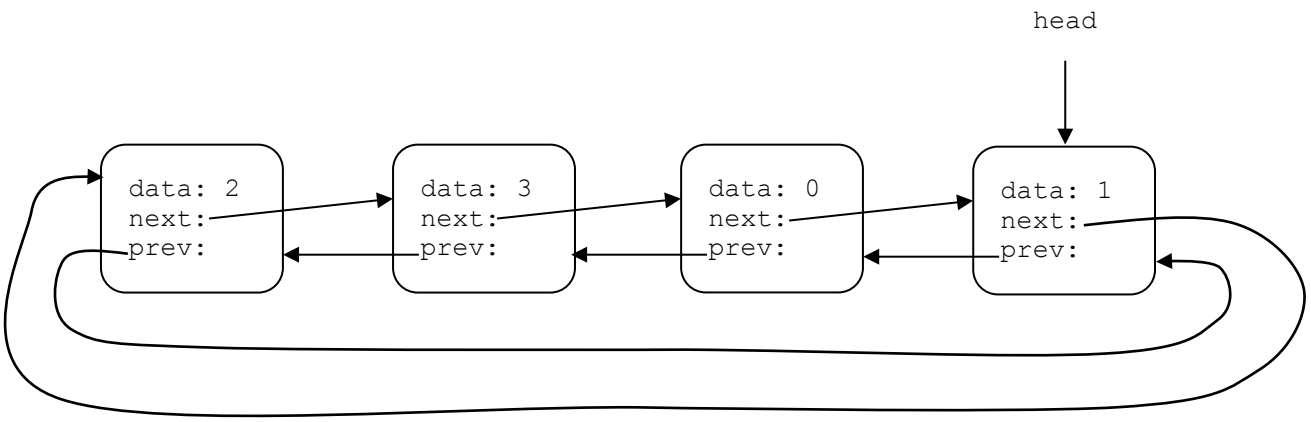



```
elemPtr removeElem(elemPtr ptr)
{
  ptr->prev->next = ptr->next;
  ptr->next->prev = ptr->prev;
  ptr->next = ptr->prev = ptr;
  return ptr;
}
```

d) 3p Modify the code so that the elements of the list printed by the call to `printList()` is printed in the exact order as they (the numbers) where entered to `main()`. You may not modify the code of the function `printList()`.

Answer: What we want to achieve is that the elements in the list is in FIFO-order. We achieve this by inserting new elements last in the list, that is at the position pointed to by `head->prev`. That is we will build the list to look like this for the input: 1, 2, 3, 0

Note: If the test code (main) is not flawed, and it is not flawed in this case, one should NOT try to adapt the test code to the peculiar abstract data type implementation to get the desired results. The proper thing to do is to correct the abstract data type.



Thus we change `insertElem()` to do the following:

```
void insertElem(const elemPtr point, elemPtr newPtr){
  newPtr->prev = point->prev;
  newPtr->next = point;
  point->prev->next = newPtr;
  point->prev = newPtr;
}
```

Code given for question 7:

```
typedef struct x {
  int data;
  struct x *prev, *next;
} elem, *elemPtr;

elemPtr newElem(int data){
  elemPtr tmp = (elemPtr) malloc(sizeof(elem));
  tmp->data = data;
  tmp->prev = tmp->next = tmp;
  return tmp;
}

void insertElem(const elemPtr point, elemPtr newPtr){
  newPtr->prev = point;
  newPtr->next = point->next;
  point->next->prev = newPtr;
  point->next = newPtr;
}

void printList(const elemPtr head) {  // DO NOT MODIFY THIS CODE
  elemPtr p = head->next;
  printElem(head);
  while(p != head) {
   printElem(p);
   p = p->next;
}}

elemPtr removeElem(elemPtr ptr) {
 /* ADD CODE */}

int main(){                    // DO NOT MODIFY MAIN()
  int tal;
  elemPtr head = NULL, ptr = NULL;
  printf("GIVE A NUMBER: ");
  scanf("%d", &tal);
  head = newElem(tal);
  while(tal != 0) {
    printf("GIVE A NUMBER: ");
    scanf("%d", &tal);
    insertElem(head, newElem(tal));
  }
  printList(head);
}
```

Question 8:

Which sorting algorithms are the best to use for the following examples? Motivate why (one/two sentences per sub-question should be sufficient) and state the time complexity for the worst case and the expected.

a) 2.5p Small one-dimensional array with less than 50 elements

Answer: For a small array, we assume the average size would be c:a 25 elements, Insertion sort is likely to be the fastest sorting algorithm (and it is stable) 2.5p. Merge- or Quicksort with cut-off to Insertion sort 2p, Merge- or Quicksort without cut-off 1p.

b) 2.5p Large one-dimensional array, memory usage is critical

Answer: If we are not allowed to use any extra memory at all we should use Heapsort while if we can use an in-place algorithm which uses log(N) extra memory but is faster we should use Quicksort 2.5p. Only suggesting Heapsort based on that it does not use extra memory 2p. Only suggesting Quick-sort as it is in-place 2p.

c) 2.5p Large one-dimensional array, memory usage is not critical, the relative order of elements with the same key should be preserved

Answer: Mergesort is the fastest stable sorting algorithm for large input 2.5p.

d) 2.5p Large one-dimensional array with few inversions

Answer: Given few inversions Insertion sort will sort the array in near linear time 2.5p. Merge or Quick sort with cut-off to Insertion sort 1.5p. Merge- or Quicksort without mentioning cut-off 1p.

Question 9:

Draw and explain in detail how the data structures below are built for the input below. Assume that comparisons are alphabetic.

a) 3p A binary search tree

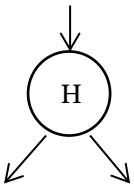
b) 3p A hash table where the hash returns `ASCIIcode(bokstav) - ASCIIcode(A)`

c) 4p A ”left leaning red black tree”

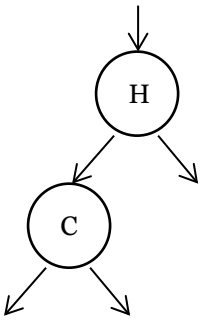
Input: **`H C B A D E`**

Answer a):

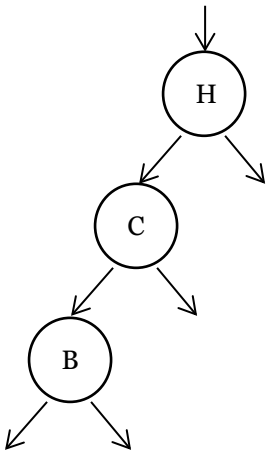
Insert H: creates a one node tree



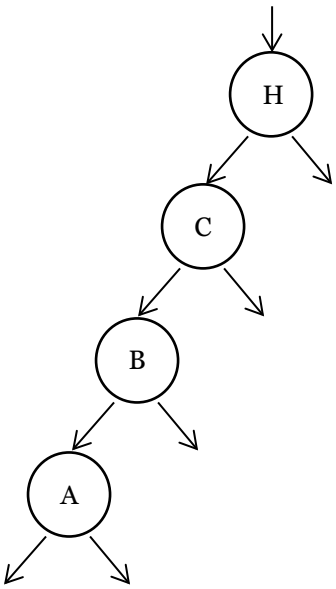
Insert C: C is less than H, insert to the left



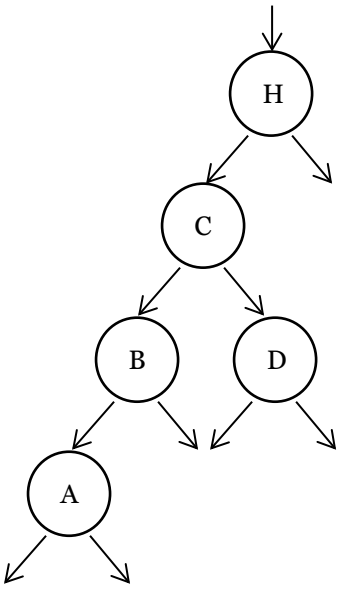
Insert B: B is less than H and less than C, insert to the left of C



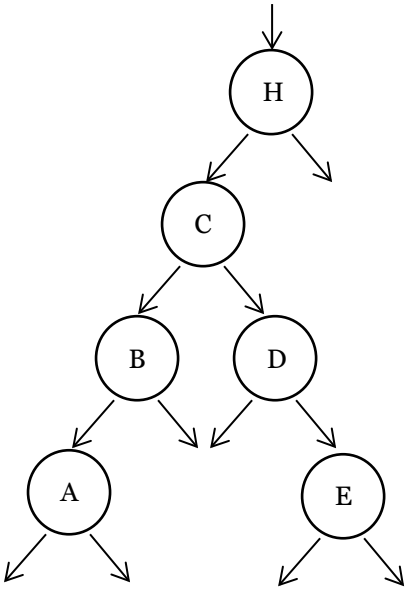
Insert A: A is less than H, less than C and less than B, insert to the left of B



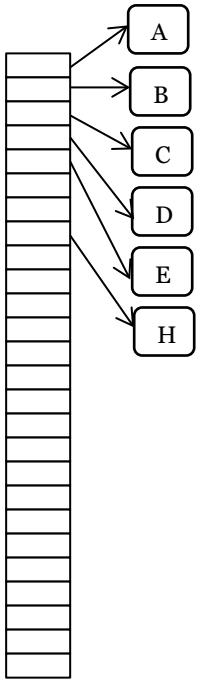
Insert D: D is less than H – go left, D is greater than C. Insert to the right of C



Insert E: E is less than H, greater than C and greater than D. Insert to the right of D

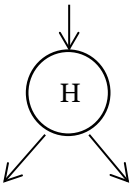


Answer b): In this case the answer depends on if you show the result for linear probing or separate chaining. In the following answer we use separate chaining. The hash (n.b. not the hashCode) falls in the range [0, 25]. Thus we know that the hash table array has 26 elements.

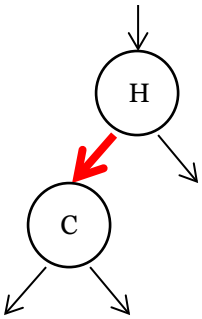


Answer c):

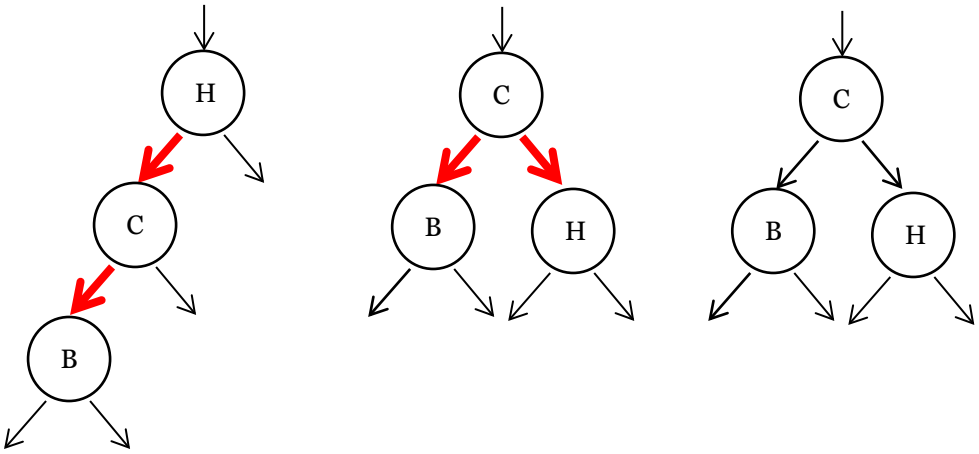
Insert H: Create a node colored red containing H. As it will be the root color it black.



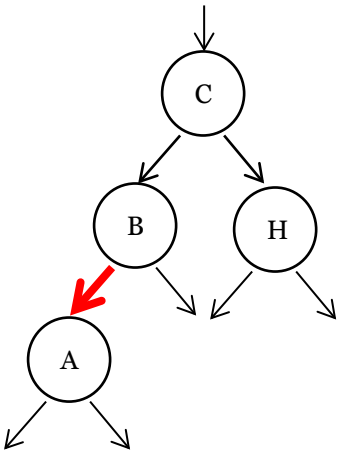
Insert C: Go left from H. Create a new node containing C insert it to the left of H with a red link to it.



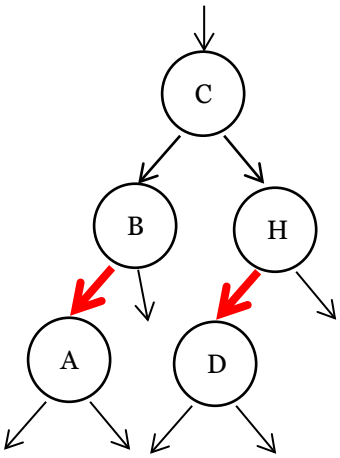
Insert B: B is less than H and less than C. 1) Insert a new node with a red link from C. 2) Two red left links in a row – rotate right. 3) Both children red – flip colors to black



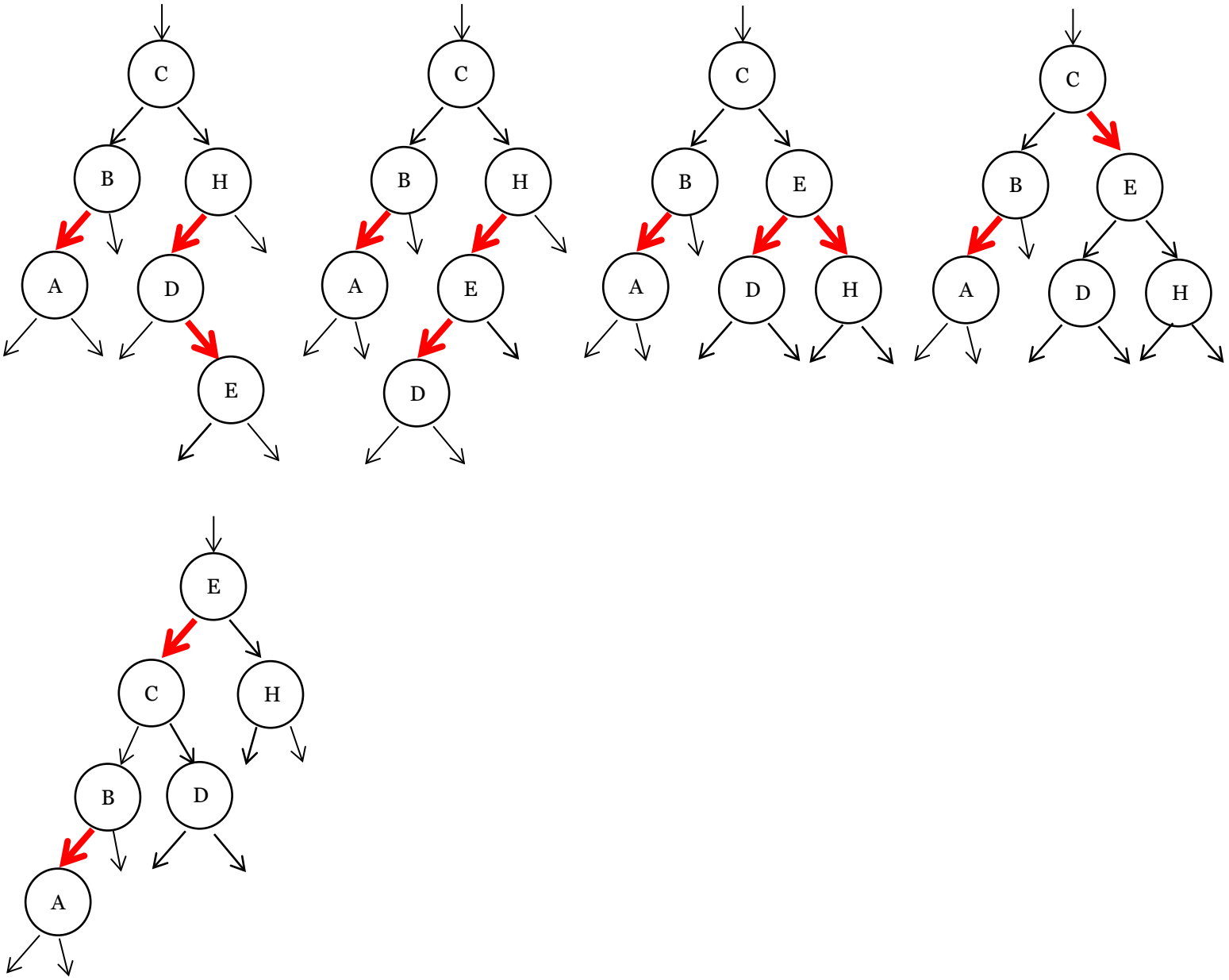
Insert A: A is less than C and less than B. Insert a new node to the left of B with a red link to it



Insert D: D is greater than C – go right, D is less than H. Insert a new node to the left of H with a red link to it.



Insert E: E is greater than C – go right, less than H – go left, greater than D.1) Insert a new node to the right of D with a red link to it. 2) Two consecutive red links – rotate left. 3) Two red links in a row – rotate right. 4) Two red children – flip colors to black 5) Red right link – rotate left



Question 10:

Assume a graph with weighted edges. Is it possible to use Breadth-First Search (BFS) as defined for graphs with unweighted edges with some simple modification to calculate shortest paths (edge weights are interpreted as the length of the edges)? If it is possible explain how the algorithm should be modified! If not, explain why it is not possible!

Answer: BFS can be used without modifications to find shortest paths iff all weights are the same. However, BFS assumes that weights are non-negative and it adds edges based on the shortest distance from the source as the distance from the source is equal to the number of edges from the source in a non-weighted graph. This is the same idea as for Dijkstra's algorithm. In BFS we use a FIFO queue and insert nodes to visit in increasing order of their distance from the source. In effect this is similar to using a priority queue where the priority is the (implicit) distance from the source (i.e. number of edges). For Dijkstra's algorithm we use the distance explicitly as priority and relax edges.

Part 2: Higher grades

The following questions are for grades A-D. Complete solutions where every part has a motivation is required. For the programming assignments you may write pseudo-code and use the algorithms and data structures covered in the course. In such case be careful to describe which algorithms/data structures/methods you use!

Question 11:

Assume we have a simple data base of persons in an array. Each person has the following attributes:

- First name
- Family name
- Year of birth

Assignment:

a) Write/describe data types and a sorting method for the person data and a method to sort the array according to: Family name, Year of birth, First name in ascending priority order according to these three attributes. Show that your solution is correct.

b) Calculate the time and memory complexity of the method

To obtain maximum points the method should be as efficient as possible with regard to the time and memory complexity.

Example order:

Andersson, 1965, Anton

Andersson, 1968, Alma

Bengtsson, 1954, Sture

Answer a): Outline: Define a data type (class) for persons which implements the Comparable interface. Then sort the data by Quicksort if there are no requirements on stability or Mergesort if stability is required.

```
public class Person implements Comparable<Person>
{
  private final String firstName;
  private final String familyName;
  private final int yearOfBirth;

  public Person(String firstN, String lastN, int year)
  //should check that year is legal
  { firstName = firstN; lastName = lastN; yearOfBirth = year; }

  public int compareTo(Person other)
  {
    // should do if(other == null) throw suitable exception
    if(this == other) return 0;
    if(this.familyName > other.familyName) return +1;
    if(this.familyName < other.familyName) return -1;
    if(this.yearOfBirth > other.yearOfBirth) return +1;
    if(this.yearOfBirth < other.yearOfBirth) return -1;
```

```
    if(this.firstName > other.firstName) return +1;
    if(this.firstName < other.firstName) return -1;
    return 0;
  }
}
```

Note: Some students have suggested that the array could be sorted three times by a stable sorting algorithm. First it is would be sorted by the last name, then by year of birth and finally by the first name. While this would solve the problem of sorting the data correctly it is a poor idea. 1) It increases the time to sort the data by a factor of three which is not efficient. 2) As the API of the sorting methods in JAVA assumes that the objects sorted implements the comparable interface it makes it hard to use these methods without substantially complicating the code.

Answer b): Assume that we can use our data type for persons for the array (if not we would have to create an auxiliary array of N person objects). Then the time and memory complexity is determined by the sorting method. The execution time is O(Nlog(N)) for both Quicksort and Mergesort. Quicksort uses O(log(N)) extra memory while Mergesort uses O(N) extra memory.

Question 12:

Describe/write code for data structures and methods for a priority queue which allows the user to change priorities of elements in the queue. The priority queue should always return the element with the highest priority when `dequeue()` is called.

a) Motivate your choice of data structure/algorithm that you use as a basis for your implementation

b) Describe the method to change the priority of an element in the queue.

c) Show that the priority queue is correct given your modifications

d) Calculate the time complexity for the method to change the priority of an element in the queue.

Answer: There is not a single way to solve this question. Hence, we outline the answer.

a) If O(log(N)) performance is needed then one could base the implementation on a heap. For simplicity it may be easier to start with a simpler data structure such as a linked list though time complexity will be O(N)
b) The simplest way of implementing this would be to find the element in the queue, remove it from the queue and then insert it again with the new priority.
c) If the functionality is implemented as suggested in b) and we based our implementation on a correct priority queue implementation, then it is sufficient to show that the remove operation results in that the remaining elements in the priority queue are correctly ordered according to the priority queue algorithm chosen in a)
d) The time complexity in big-Oh notation would be the larger of the big-Oh time complexity to find the element in the queue and the time complexity to insert the element into the queue

Question 13:

Assume that we have a simple data base of individuals in a social network. The data representation for each individual in the network is that there is a list of other persons in the network that the individual is a follower of associated with the individual. Assume each individual has a unique identifier. We define a ***closed group of followers*** as a group in which you can get back to any individual you start from by following a chain of followers (A follows B who follows C who follows A)

a) Describe data structures and methods which allows us to identify all closed groups of followers in the data base.

b) Describe the time complexity to find the groups.

Answer a): The problem is modeled as the problem of finding all strongly connected components in a directed graph where persons are vertices and that a person A follows another person B is a directed edge from node A to node B. The algorithm used to compute the strongly connected components is the Kosaraju-Sharir algorithm. For details see pages 584-591

Answer b): The Kosaraju-Sharir algorithm identifies the strong components in time and memory complexity proportional to E+V (the number of edges plus the number of vertices/nodes)

Question 14:

Given a one-dimensional array of length N where the elements are floating point numbers uniformly distributed over the interval `[0,n]` where `n <= N-2`.

a) Describe a method to sort the elements in the array which has near `O(1)` time complexity. Show how you calculate the time complexity.

b) Compare the expected execution time of your algorithm to some of the fast comparison based sorting methods we have covered in the course. (Hint: use tilde-notation when you compare, you need not use exact numbers for constants)

Answer a): There are strong similarities between this problem and that of a hash-table. The method is:

1) Create an array of size N of priority queues

2) Scan the array of floating point numbers and create a "hash" from the floating point number ***f*** by taking the integer part of (***f****(N-1))/n. It is necessary to scale the floating point numbers to cover the whole range [0,N-1] to ensure that each priority queue (bin) only will have a small number of elements. Use this integer part as an index into the array of priority queues. Insert <key, priority> into that priority queue using the floating point number as both key and priority.

3) Scan the array of priority queues and dequeue the elements of the priority queues keeping track of the number of elements dequeued, we call this ***k,*** and update the array of floating point numbers element with index ***k-1*** with the key of the element dequeued (or the priority) i.e. the floating point number.

4) As we scale the uniformly distributed floating point numbers to create hash values which are uniformly distributed over the interval [0,N-1] we can assume that the maximum number of elements in any of the priority queues will be limited by a small constant number.

   The pre-processing time to create the array of priority queues will be O(N). The sorting is done by: scanning the original N elements of the input array, computing a "hash" in constant time for each floating point number and inserting it into a priority queue in constant time. Then scanning the array of N priority queues and performing a bounded number of constant time operations on each priority queue to empty it. That is all operations are O(N) – hence the array would be sorted in O(N) time using O(N) extra memory for the array of priority queues.

Answer b): A standard comparison based sorting method to sort a large number of floating point numbers would be Quicksort. Quicksort is ~***a***Nlog(N) where a is a constant. The method proposed above is ~***b***N. Quicksort would thus be faster if ***a***log(N) < ***b*** We can assume that ***a*** is a relatively small number while ***b*** involves creating the array of priority queues, computing hashes, inserting and removing elements from the priority queues. Thus ***b*** could well be larger than ***a***log(N) for many applications.