CORE EXAMINATION
Department of Computer Science
New York University
May 16, 1997

SOLUTIONS

Programming Languages and Compilers

Question 1

Consider the following two programs whose purpose is two swap the values of two variables:

```
void swap (int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

void swap (int x, int y) {
   int foo () {
      int temp = x;
      x = y;
      return temp;
   }
   y = foo();
}
```

Disregarding the C-syntax, assume that calls to these procedures pass parameters a) strictly by value, b) by reference, c) by name. In each case, discuss whether the procedures correctly swap the values. You may want to consider different forms that the variables can take.

Solution

- No, Yes, and No. Parameter passing by value, as used e.g. in C, can never affect the actuals in the call. For parameter passing by name, consider the call swap (i, A[i]).

- Also No, Yes, and No. For parameter passing by name, the previous example works, because the call to foo captures the value of A[i], but consider the call swap(i, A[i++]).

Question 2

In the language of your choice (but a real language, not pseudocode) give the specification for a queue manager. This specification may be in the form of a package, a module, a class, or some other kind of software component. A queue is characterized by the type of the objects that it can hold, and by its maximum capacity. The queue supports at least operations of insertion, removal, and test for emptiness. You are to write the implementation of at least one of these operations. Explain carefully how your code addresses the following:

1. Using the same specification, how can one create queues that hold values of different types? How does one create queues with different capacities?
2. What exceptional conditions can arise when executing each queue operation?
3. How are these conditions signalled to the caller?

Finally, in your chosen language, write a driver program that tries to overfill a queue and handles the corresponding exceptional condition.

Solution

The question asks for a software component that encapsulates the queue manager. In Ada this will be a package, in C++ a class, and in Java a class or a package with several classes. In C, there is no clear notion of a software component in this sense. A group of functions using some common types does not constitute a clean abstraction, and the only (imperfect) way of indicating that a collection of functions has some unified meaning is to place them in a file by themselves. This is clearly inferior to the other solutions. A possible solution in Ada is given by:

```
generic
   type Element is private;
   Capacity: Natural;
package Queue_Manager is
   type Queue is private;
   procedure Insert  (Q: in out Queue; E: Element);
   procedure Remove  (Q: in out Queue; E: Element);
   function Is_Empty (Q : Queue) return Boolean;
   function Front    (Q : Queue) return Element;

   Empty_Error, Overflow_Error: Exception;
private
   type Arr is array (Natural range <>) of Element;

   type queue is record
      Front, Back: Natural := 1;
      Num  : Natural := 0;
      Contents :  Arr (1 .. Capacity);
   end record;
end Queue_Manager;
```

The algorithms for each subprogram can be found in any textbook.

- In order to parametrize the queue by the type of the element stored in it, we want to use a generic package (in Ada) or a template (in C++). C does not have a generic facility, so the C solution has to use void* pointers, which allows the construction of completely heterogeneous queues, but loses all type safety. Java does not have a generic capability either. Declaring the element type to be Object provides the same universal polymorphism as the C solution, and the corresponding loss of type checking (except that the run-time will provide type-checking when retrieving from such a queue.

- The parametrization by size can be obtained in various ways. In Ada, the generic package may contain an integer parameter that specifies the capacity of a queue, or the queue type may be discriminated. Similarly, the C++ template may have an integer parameter, or the constructor may have one. In C and Java, the call to the constructor will specify size.

- Attempting to insert an element into a full container, or attempting to delete from an empty container, are the standard illegal operations that we must be able to handle. The cleanest is to define exceptions that signal the error, and raise these exceptions in the queue operations, when they cannot be performed. In C, one must use the low-level setjmp-longjmp mechanism (using a return value is conventional but clearly inferior because every direct or indirect caller must examine the return value). In C++ and Java, exceptions are classes, In Ada they are special entities, but the mechanism is similar: there is a mechanism to signal an exception (*raise* or *throw*) and a way to handle an exception that has been signalled (use an exception handler or a *try-catch* block. For example, overflow is detected as follows:

```
procedure Insert (Q : in out Q; E : Element) is
begin
   if Num = Capacity then raise Overflow_Error;
   else ...     --  modify Num and Back index, and insert element.
   end if;
end Insert;
```

  and the exception is used as follows:

```
declare
   package Int_Queue is new Queue_Manager (Integer, 100);
   use Int_Queue;
   Test_queue : Queue;
begin
   for I in 1 .. 1000 loop
      Insert (Test_Queue, I**2);
   end loop;
exception
   when Overflow_Error =>
      Put_Line ("Queue capacity exceeded. Program terminated");
      raise Program_error;                        -- for example.
end;
```

Question 3

Consider the following in a language allowing nested procedures

```
procedure a is
    x : integer;

    procedure b is
        y : integer;

        procedure c is
          z : integer;
        begin
           print (x + y + z);
        ..
        end c;
    ..
    end b;
..
end a;
```

The two most common methods of handling the evaluation of x+y+z in procedure c are referred to as the *global display* method and the *static chain* method. Answer the following questions about these methods (a few sentences for each will be sufficient).

1. What is a global display?
2. What is the state of the global display at the point of evaluating x+y+z?
3. How is the global display used to evaluate x+y+z?
4. How is the global display maintained?
5. What is a static chain?
6. What is the state of the static chain at the point of evaluating x+y+z?
7. How is the static chain used to evaluate x+y+z?
8. How is the static chain maintained?
9. The presence of multi-tasking in a language makes the static chain method more attractive, why?
10. What is meant by register allocation in a compiler? If you have a really good register allocator, the static chain method becomes relatively more attractive, why?
11. The presence of procedure pointers in a language makes the static chain method more attractive, why?

Solution

1. A global display is a global data struture that is an array where the N'th element is a reference to the stack frame of the currently visible N'th level procedure.

2. In this case, the first three entries in this global display at the point where x+y+z is evaluated would be pointers to the stack frames of a, b, and c respectively.

3. Evaluation requires one level of indirection, to first get the right frame pointer, then an offset reference to get the right variable. The indirection can be eliminated by keeping the display in registers.

4. The display is maintained by adjusting the appropriate display entry on calling a procedure, and restoring it on return.

5. The static chain method works by keeping in each stack frame a constant that is a reference to the stack frame of the statically enclosing procedure.

6. In this case, at the point where x+y+z is evaluated, the stack frame of c would contain a reference to the stack frame of b, which would in turn contain a reference to the stack frame of a.

7. The evaluation involves climbing this chain to the right level (in this example, z is immediately available, y is obtained by using the static link in the c frame, x is obtained by using the static link in the b frame, which in turn requires using the static link in the c frame.

8. The static chain is maintained by passing an implicit parameter, which is a reference to the stack frame of the procedure that statically encloses the called procedure.

9. In a multi-tasking environment, one would need a global display for each task, and some way of switching between them. But the static chain requires no special handling, since everything is local to stack frames, which are task specific in any case.

10. Register allocation refers to the phase in a compiler that chooses what values to put and keep in registers, attempting to minimize memory references. The static chain method may require repeated references high up in the chain, but a good register allocator may be able to arrange to get the link once, and then keep it in a register, reducing the penalty from this situation.

11. A procedure pointer in a language with nested procedures is more than a code pointer, since one must also reference the enclosing environment. In the static chain case, it is good enough to just reference the stack frame of the relevant procedure, since the chain link is in this frame. For the global display method, a procedure pointer must contain a complete copy of the display.

Question 1

The well-known *Belady's anomaly* shows that for some page replacement algorithms, the number of page faults can increase if the number of frames is greater.

- Briefly describe why the LRU page replacement algorithm does not suffer from Belady's anomaly. LRU is considered to be a good page replacement algorithm, but it is not used in practice. Explain why.

- briefly describe the following, so called *LRU approximation algorithms*:
(a) the additional-reference-bits algorithm, and (b) the second-chance algorithm.

Solution
In the LRU algorithm, if the number of frames is increased, the set of pages actually kept in memory becomes a superset of the original set. Therefore no new page faults can appear and some old ones can disappear. However, the algorithm is too expensive in its pure form to be used in a real OS.

Additional-reference-bits algorithms maintain information as follows. For some selected number of contiguous time intervals in the past, we know whether a particular page was accessed in a particular time interval. The approximation for "least recently used page" is : page which is not used for the largest number of contiguous time intervals from the present into the past.

In the second chance algorithm, a FIFO queue is maintained. If a page at the head of the queue has a reference bit of 0, it is replaced. It is a reference bit of 1, the reference bit is set to 0 and the page is moved to the end of the queue.

Question 2

Briefly describe and compare the following file allocation methods: (a) Contiguous allocation, (b) linked allocation,and (c) indexed allocation (simplest possible variant).

Your comparison should address: (i) efficient sequential and direct access, (ii) wasted space, and (iii) dynamically growing files.

Solution

Contiguous allocation requires that the file occupy a sequence of contiguous blocks on the disk. This yields very cast sequential and direct access. Any reasonable implementation for dynamically growing files will waste considerable amount of space or will be very inefficient because of reallocation.

Linked allocation maintains the file as a linked list. This does not waste much space (links are the only overhead). This yields fast sequential access but slow direct access.

Indexed allocation maintains the file as a set of blocks pointed at by an index table storing pointers to the blocks of the file. This is space efficient, unless the file as a very small number of blocks. Both sequential and direct access are reasonably fast.

QUESTION 1.

For each of the following statements state whether it is true or false, and give one sentence explaining why or give a counterexample.

1. Consider Dijkstra's algorithm on a graph G = (V,E) with positive weights. Let S be a subset of V, S = set of vertices whose minimum distance from the source is already known. At some point in the algorithm let v be a new vertex just added to S, and let $w \notin S$. Is it possible that the new shortest special path from the source to w should first visit v, and then some other node y in S?

2. Dr. Flubberty has invented a new comparison based algorithm for sorting arbitrary numbers that runs in time $O(n \cdot \log(\log n))$. Dr. Jibberty claims it cannot be correct. Could it be?

3. In an undirected graph, every cycle must have at least one vertex that is an articulation point, that is, a vertex whose removal would cause the graph to be disconnected.

4. $16^{\frac{\log n}{2}}$ is the same order as $5^{\log n}$.

5. Let G be a directed graph with positive weights. If P is the least cost path from v to w and Q is the least cost path from w to u, then P followed by Q is the shortest path from v to u.

6. Suppose we have an array A of length N of integers in the range $[0, .., N^3 - 1]$. The numbers can be sorted in linear time, but it is not possible to do this in less than $O(N^3)$ storage.

7. In a breadth first search of an undirected graph, there are no back edges and no forward edges.

8. Dijkstras's algorithm to find the cheapest paths from a source to all vertices still works in all cases if there are negative edge weights, but no negative cost cycles. (A negative cost cycle is a cycle whose total cost is negative).

9. Prim's algorithm to find the Minimum Spanning tree of an undirected graph still works in all cases if the edges are allowed to have negative weights and negative cost cycles are allowed too.

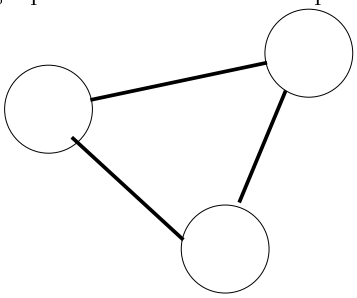
10. Suppose you are given a minimum spanning tree T of a graph G with positive weights on the edges. If we add the vertex v to the graph G, along with some positive weight edges from v to vertices of G, then the weight of a minimum spanning tree of the new graph is always bigger than the weight of T.

SOLUTION 1.

1. The new vertex can not be part of a minimum path from v to$w$ through y. Since v was just added, but y is already in S, using v must provide a longer path than the one previously computed that goes to y, and would also be a more expensive way to get to w.

2. Decision tree analysis of a comparison based sort gives a lower bound running time of $O(n \log n)$, so an $O(n \log \log n)$ must not be correct. (Dr. Jibberty is right).

3. False. Every cycle does not necessarily have an articulation point. The simple cycle below has no articulation point. A complete graph has no articulation point.



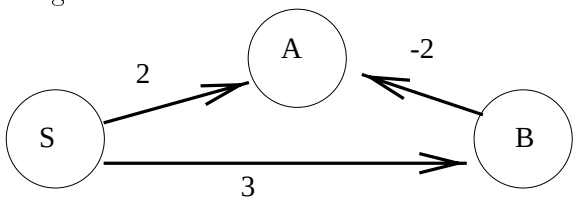
4. False. $16^{\log n/2} = 4^{\log n} < 5^{\log n}$. Since $lim_{n \to \infty} \frac{4^{\log n}}{5^{\log n}} = 0$, they are not the same order as each other.

5. False. For example, there could be a direct edge that costs less than P and/or Q that connect v to u.

6. False. The array A could be sorted using 3 passes of radix sort, which uses $O(N)$ storage. (In other words, treat the numbers as 3 digit number in base N.)

7. True. A breadth first search in an undirected graph would reach a vertex that is adjacent to it as it searched the first level of adjacent vertices, so that it would already be seen using a tree edge and not a back edge. The same argument eliminates forward edges.

8. False. Dijkstra's algorithm does not work if there are negative edges weight either, since there might be a cheaper path found later that would need to undo a previous step. For example, in the graph below A would be chosen as having a minimum path of length 2 from the source S. However, a cheaper path exists through B.



9. True. Prims' algorithm works in all cases, whether or not there are negative edges or cycle. (In fact, if a positive constant c is added to all edges, they are processed in the same order, and the cost of the new tree is $(N-1) \cdot c$ more than the cost of the old tree.)

10. False. The new vertex v could have a cheaper connection to a previously existing vertex than the one in the old MST.

QUESTION 2.

Let $G = (V, E)$ be a directed acyclic graph, represented by adjacency lists. Give a linear time algorithm to find the longest path in a dag. You can use pseudo-code, as long as the algorithm is presented clearly. (Hint: use DFS; what information is available when DFS(v) completes?).

SOLUTION 2.

The idea is to do a depth first search of the dag, and for each vertex, keep track of the longest path starting at that vertex. This can be done in a postorder fashion, once the longest paths reachable from the adjacent vertices are computed.

Thus, one pass through all the edges is still sufficient, and the running time remains the usual depth first search time of $O(V + E)$.

```
procedure DFS (V:vertex);
begin
  Visited[V] := true;
  MaxPath[V] := 0;
  For each W adjacent to V do
    begin
      if not visited[W] then DFS(W);
      MaxPath[V] = max(MaxPath[V], 1 + MaxPath[W]);
    end

end
```

QUESTION 3.

Suppose that you have a heap whose keys are integers (positive or negative); the item with maximum key is at the root. The problem is, given a positive integer k, to determine whether there are at least k positive keys in the heap. Design an algorithm that solves the problem in time $O(k)$, regardless of the size of the heap. Be sure to argue that the running time is $O(k)$.

SOLUTION 3.

The idea is to traverse the heap, counting the number of positive keys. If we encounter an item whose key is nonpositive, then we can ignore all items below it, because they must have nonpositive keys also (by the order property of heaps). If our count ever reaches k, then we can stop immediately.

More precisely, we will maintain a global integer variable called `count`, which is initialized to 0. Then we call the following recursive procedure, with `root` set to the root of our heap.

```
procedure traverse(root);
begin
  if (count < k) and (root <> nil) and (root^.key > 0) then
    begin
      count := count + 1;
      traverse(root^.left);
      traverse(root^.right)
    end
end
```

After calling this procedure, if `count` $\geq k$, then there are at least k positive keys; otherwise, there are not.

What is the running time of this algorithm? We have to bound the number of items that are visited. Let S be the set of visited items whose children also get visited. Because `count` is incremented for each such item, we must have $|S| \leq k$. Because the only items visited are the items in S and their children, at most $2k + 1$ items are visited. Hence the running time is $O(k)$.

QUESTION 1.

For each of the following statements state whether it is true or false, and give one sentence explaining why or give a counterexample.

1. Consider Dijkstra's algorithm on a graph G = (V,E) with positive weights. Let S be a subset of V, S = set of vertices whose minimum distance from the source is already known. At some point in the algorithm let v be a new vertex just added to S, and let $w \notin S$. Is it possible that the new shortest special path from the source to w should first visit v, and then some other node y in S?

2. Dr. Flubberty has invented a new comparison based algorithm for sorting arbitrary numbers that runs in time $O(n \cdot \log(\log n))$. Dr. Jibberty claims it cannot be correct. Could it be?

3. In an undirected graph, every cycle must have at least one vertex that is an articulation point, that is, a vertex whose removal would cause the graph to be disconnected.

4. $16^{\frac{\log n}{2}}$ is the same order as $5^{\log n}$.

5. Let G be a directed graph with positive weights. If P is the least cost path from v to w and Q is the least cost path from w to u, then P followed by Q is the shortest path from v to u.

6. Suppose we have an array A of length N of integers in the range $[0, .., N^3 - 1]$. The numbers can be sorted in linear time, but it is not possible to do this in less than $O(N^3)$ storage.

7. In a breadth first search of an undirected graph, there are no back edges and no forward edges.

8. Dijkstras's algorithm to find the cheapest paths from a source to all vertices still works in all cases if there are negative edge weights, but no negative cost cycles. (A negative cost cycle is a cycle whose total cost is negative).

9. Prim's algorithm to find the Minimum Spanning tree of an undirected graph still works in all cases if the edges are allowed to have negative weights and negative cost cycles are allowed too.

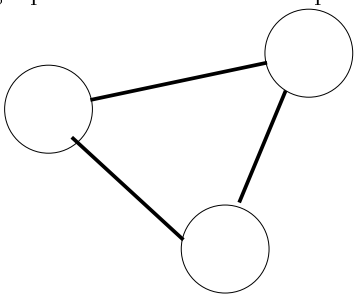
10. Suppose you are given a minimum spanning tree T of a graph G with positive weights on the edges. If we add the vertex v to the graph G, along with some positive weight edges from v to vertices of G, then the weight of a minimum spanning tree of the new graph is always bigger than the weight of T.

SOLUTION 1.

1. The new vertex can not be part of a minimum path from v to$w$ through y. Since v was just added, but y is already in S, using v must provide a longer path than the one previously computed that goes to y, and would also be a more expensive way to get to w.

2. Decision tree analysis of a comparison based sort gives a lower bound running time of $O(n \log n)$, so an $O(n \log \log n)$ must not be correct. (Dr. Jibberty is right).

3. False. Every cycle does not necessarily have an articulation point. The simple cycle below has no articulation point. A complete graph has no articulation point.



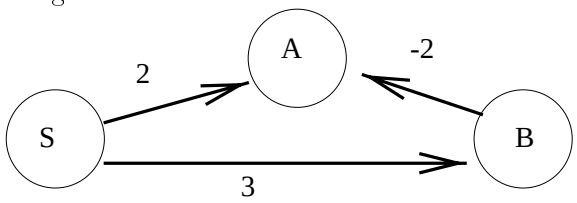
4. False. $16^{\log n/2} = 4^{\log n} < 5^{\log n}$. Since $lim_{n \to \infty} \frac{4^{\log n}}{5^{\log n}} = 0$, they are not the same order as each other.

5. False. For example, there could be a direct edge that costs less than P and/or Q that connect v to u.

6. False. The array A could be sorted using 3 passes of radix sort, which uses $O(N)$ storage. (In other words, treat the numbers as 3 digit number in base N.)

7. True. A breadth first search in an undirected graph would reach a vertex that is adjacent to it as it searched the first level of adjacent vertices, so that it would already be seen using a tree edge and not a back edge. The same argument eliminates forward edges.

8. False. Dijkstra's algorithm does not work if there are negative edges weight either, since there might be a cheaper path found later that would need to undo a previous step. For example, in the graph below A would be chosen as having a minimum path of length 2 from the source S. However, a cheaper path exists through B.



9. True. Prims' algorithm works in all cases, whether or not there are negative edges or cycle. (In fact, if a positive constant c is added to all edges, they are processed in the same order, and the cost of the new tree is $(N-1) \cdot c$ more than the cost of the old tree.)

10. False. The new vertex v could have a cheaper connection to a previously existing vertex than the one in the old MST.

QUESTION 2.

Let $G = (V, E)$ be a directed acyclic graph, represented by adjacency lists. Give a linear time algorithm to find the longest path in a dag. You can use pseudo-code, as long as the algorithm is presented clearly. (Hint: use DFS; what information is available when DFS(v) completes?).

SOLUTION 2.

The idea is to do a depth first search of the dag, and for each vertex, keep track of the longest path starting at that vertex. This can be done in a postorder fashion, once the longest paths reachable from the adjacent vertices are computed.

Thus, one pass through all the edges is still sufficient, and the running time remains the usual depth first search time of $O(V + E)$.

```
procedure DFS (V:vertex);
begin
  Visited[V] := true;
  MaxPath[V] := 0;
  For each W adjacent to V do
    begin
      if not visited[W] then DFS(W);
      MaxPath[V] = max(MaxPath[V], 1 + MaxPath[W]);
    end

end
```

QUESTION 3.

Suppose that you have a heap whose keys are integers (positive or negative); the item with maximum key is at the root. The problem is, given a positive integer k, to determine whether there are at least k positive keys in the heap. Design an algorithm that solves the problem in time $O(k)$, regardless of the size of the heap. Be sure to argue that the running time is $O(k)$.

SOLUTION 3.

The idea is to traverse the heap, counting the number of positive keys. If we encounter an item whose key is nonpositive, then we can ignore all items below it, because they must have nonpositive keys also (by the order property of heaps). If our count ever reaches k, then we can stop immediately.

More precisely, we will maintain a global integer variable called `count`, which is initialized to 0. Then we call the following recursive procedure, with `root` set to the root of our heap.

```
procedure traverse(root);
begin
  if (count < k) and (root <> nil) and (root^.key > 0) then
    begin
      count := count + 1;
      traverse(root^.left);
      traverse(root^.right)
    end
end
```

After calling this procedure, if `count` $\geq k$, then there are at least k positive keys; otherwise, there are not.

What is the running time of this algorithm? We have to bound the number of items that are visited. Let S be the set of visited items whose children also get visited. Because `count` is incremented for each such item, we must have $|S| \leq k$. Because the only items visited are the items in S and their children, at most $2k + 1$ items are visited. Hence the running time is $O(k)$.