NANYANG TECHNOLOGICAL UNIVERSITY

SEMESTER 2 EXAMINATION 2020-2021

EE2008 – DATA STRUCTURES AND ALGORITHMS

April / May 2021 Time Allowed: 2½ hour

INSTRUCTIONS

1. This paper contains 4 questions and comprises 4 pages.
2. Answer all questions.
3. All questions carry equal marks.
4. This is a closed book examination.
5. Unless specifically stated, all symbols have their usual meanings.

---

1. (a) Design a **recursive** algorithm `lgfact(n)` for computing $\log_2 n! = \lg n!$ for an integer $n \geq 1$. The implementation is based on the formula:

$$\lg m(m+1) = \lg m + \lg(m+1)$$

where $m \geq 1$.

(10 Marks)

(b) Show that the complexity of $\lg n!$ is $O(n \lg n)$.

(5 Marks)

(c) When $n \geq 4$, rewrite

$$\lg n! = \sum_{k=1}^{j-1} \lg k + \sum_{k=j}^{n} \lg k$$

where $j = \lceil n/2 \rceil$ is the ceiling of $n/2$. It is easy to see that

$$\lg n! \geq \sum_{k=j}^{n} \lg k$$

Based on this observation, show that $\lg n! = \Omega(n \lg n)$ for $n \geq 4$.

(10 Marks)

2. (a) Consider a binary tree with `root` at node 1, as shown in Figure 1. Use the algorithm `PrintNode(1)` to determine the content of the printing process. Justify the printed output in detail.

```
Algorithm PrintNode(root) {
  if (root != null) {
    PrintNode(root.right)
    PrintNode(root.left)
    print root      // output the root number
  }
}
```

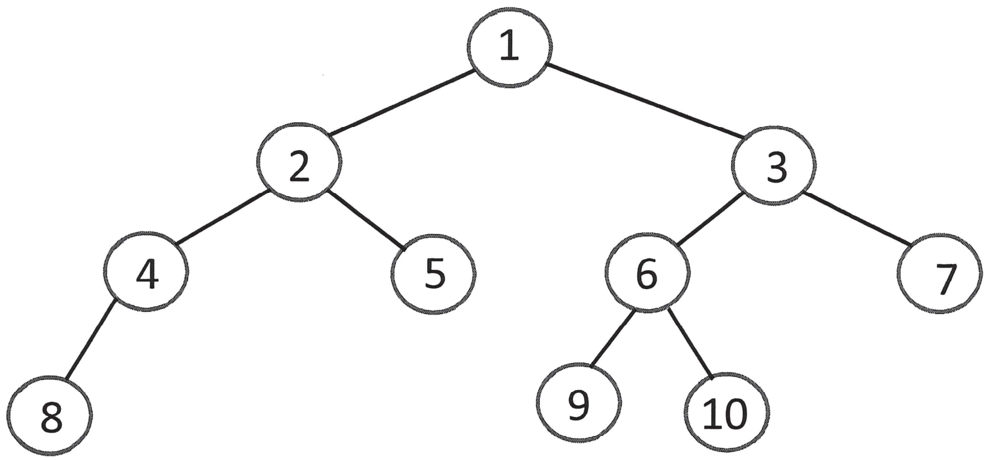


Figure 1

(10 Marks)

(b) A stack is an abstract data type (ADT) with 5 basic functions, namely, `stack_init()`, `empty()`, `push(val)`, `pop()` and `top()`, where `val` is a data item. The detailed implementation of the functions is not given for the ADT. Using the basic functions of the stack, write a function `is_double(s)` to check whether the stack `s` contains only two data items. In particular, it returns `true` if there are exactly two elements in the stack and returns `false` otherwise. Before and after the execution of `is_double(s)`, the content of the stack `s` should remain unchanged. You need to take care of the error check if necessary.

(15 Marks)

3. (a) $A[0...n\text{-}1]$ is an array representing a maxheap. Assume that a new node is inserted as $A[n]$. Write a **recursive** algorithm for implementing the siftup operation to restore the maxheap property.

(6 Marks)

(b) Let A be the following array

31	23	16	20	4	12

which is a maxheap. Now the second item that contains the value 23 is to be deleted. Explain and show each step in using the siftup and/or siftdown to recover the maxheap structure.

(6 Marks)

(c) Denote the following array as $A[1..5]$. Explain and show each step in using the mergesort to sort it.

31	23	16	20	4

(6 Marks)

(d) For two unsorted arrays A and B with m and n elements respectively (note that m may not be equal to n), to merge them into a single sorted array using the mergesort method, two different approaches are adopted: (1) mergesort A and B separately, and then merge them into a single array with necessary further sorting operations; (2) attach B to the end of A to generate a single arrary C, then conduct mergesort on C. Are the two approaches of the same complexity? Justify your answer.

(7 Marks)

4. (a) A *connected* undirected graph is represented as adjacency lists. The graph vertexes are indexed from 1 to N. Let $adj[i]$ be a reference to the first node in a linked list of nodes representing the vertices adjacent to vertex i, $i = 1,2,\ldots,N$. Write an algorithm to visit all the vertexes, starting with vertex 1, using the **depth-first search** (DFS) algorithm. Show the sequence that the vertexes are visited in an array seq$[1..N]$ in the algorithm output, where seq$[i]$ stores the index of the i-th visited vertex.

(8 Marks)

Note: Question No. 4 continues on page 4.

(b) Given a weighted graph as illustrated in Figure 2.

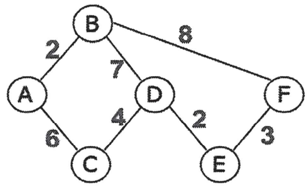


Figure 2

The Dijkstra's Algorithm is applied to find the shortest path from vertex A to all the other vertexes. Show, by illustration, each iteration of the algorithm where an edge is added to the shortest path tree (SPT) and the corresponding shortest path length that has been found.

(7 Marks)

(c) A graph is represented as an adjacency matrix A, where $A[i,j] = 1$ denotes that there is an undirected edge between vertexes i and j; and $A[i,j] = 0$ otherwise, $i,j = 1,2,\ldots,N$. Write an algorithm to read the adjacency matrix and output the corresponding adjacency lists to represent the graph.

(10 Marks)

END OF PAPER

NANYANG TECHNOLOGICAL UNIVERSITY
ELECTRICAL & ELECTRONIC ENGINEERING CLUB
PAST YEAR PAPER SOLUTION

EE2008 – DATA STRUCTURES AND ALGORITHMS SEMESTER 2 EXAMINATION 2020/21

Attempted by: Karn Watcharasupat Updated: August 29, 2021

1.
```
lgfact(n) = lg(n) + lg(n-1) + ...  + lg(2) + lg(1)
= lg(n) + lg(n-1) + ...  + lg(4) + lg(3) + lgfact(2)
= lg(n) + lg(n-1) + ...  + lg(4) + lgfact(3)
= ...
= lg(n) + lgfact(n-1)
```

(a)

```
function lgfact(n)
   if n = 1 then
      return 1
   end if
   return lg(n) + lgfact(n-1)
end function
```

(b) To prove $\lg n! = \mathcal{O}(n \lg n)$, we need $|\lg n!| \leq c(n \lg n)$ for some $c > 0$ for all $n \geq n_0$.
Fact 1: $\lg k \geq \lg l$ for all $k \geq l$ since lg is strictly increasing.

$$\begin{aligned}
\lg n! &= \lg \prod_{i=1}^{n} i && (1)\\
&= \sum_{i=1}^{n} \lg i && (2)\\
&\leq \sum_{i=1}^{n} \lg n && (3)\\
&= \lg n \sum_{i=1}^{n} 1 && (4)\\
&= n \lg n && (5)
\end{aligned}$$

Since $|\lg n!| \leq 1(n \lg n)$ for all $n \geq 1$, $\lg n! = \mathcal{O}(n \lg n)$.

(c) To prove $\lg n! = \Omega(n \lg n)$, we need $|\lg n!| \geq c(n \lg n)$ for some $c > 0$ for all $n \geq n_0$.
Fact 1: $\lg k \leq \lg l$ for all $k \leq l$ since lg is strictly increasing.
Fact 2: for $n \geq 4$, $n/2 \geq \sqrt{n}$
Fact 3: $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor \leq n/2 \leq \lceil n/2 \rceil$

$$\begin{aligned}
\lg n! &= \sum_{k=1}^{n} \lg k && (6)\\
&= \underbrace{\sum_{k=1}^{\lceil n/2 \rceil - 1} \lg k}_{\geq 0,\ n \geq 4} + \sum_{k=\lceil n/2 \rceil}^{n} \lg k && (7)\\
&\geq \sum_{k=\lceil n/2 \rceil}^{n} \lg k && (8)\\
&\geq \sum_{k=\lceil n/2 \rceil}^{n} \lg \lceil n/2 \rceil && (9)\\
&\geq (n - \lceil n/2 \rceil + 1) \lg \lceil n/2 \rceil && (10)\\
&\geq (n/2) \lg(n/2) && (11)\\
&\geq (n/2) \lg(\sqrt{n}) && (12)\\
&= \frac{1}{4} n \lg n && (13)
\end{aligned}$$

Since $|\lg n!| \geq \frac{1}{4}(n \lg n)$ for all $n \geq 4$, $\lg n! = \Omega(n \lg n)$.

2. (a) TL;DR: right-left-root traversal
   7 10 9 6 3 5 8 4 2 1

printed	trace	stack
	PrintNode(1)	->1
	PrintNode(1.right)	->1->3
	PrintNode(1.right.right)	->1->3->7
	PrintNode(1.right.right.right)	->1->3->7->null
	PrintNode(1.right.right.left)	->1->3->7->null
7	print 1.right.right	->1->3->7
		->1->3
	PrintNode(1.right.left)	->1->3->6
	PrintNode(1.right.left.right)	->1->3->6->10
	...	
10	print 1.right.left.right	
		->1->3->6
	PrintNode(1.right.left.left)	->1->3->6->9
	...	
9	print 1.right.left.left	->1->3->6->9
6	print 1.right.left	->1->3->6
3	print 1.right	->1->3
		->1
	PrintNode(1.left)	->1->2
	PrintNode(1.left.right)	->1->2->5
	...	
5	print 1.left.right	->1->2->5
		->1->2
	PrintNode(1.left.left)	->1->2->4
	PrintNode(1.left.left.right)	->1->2->4->null
		->1->2->4
	PrintNode(1.left.left.left)	->1->2->4->8
	...	
8	print 1.left.left.left	->1->2->4->8
4	print 1.left.left	->1->2->4
2	print 1.left	->1->2
1	print 1	->1
	return	<-

(b) Writer's note: in practice, you should create another stack to take care of the elements from the old stack and use proper while/for loop. But since it's only 2 elements to check here, an unrolled implementation is more easily marked **for examinations' purpose**.

In practice, please the more generalizable and cleaner version.

```
function is_double(s)
   # Exam version.  Don't use this IRL.

   if s == null then
      # null pointer
      return false # or throw error
   end if

   if s.empty() then
      # original 's' has no element
      return false
   end if

   # original 's' has at least 1 element
   # s:  [-> s1 -> ??]

   # pop 1st value from 's'
   s1 = s.pop() # s:  [-> ??]

   if s.empty() then
      # original 's' has 1 element
      # s:  [-> null]
      # push 's1' back where it was
      s.push(s1) # s:  [-> s1 -> null]
      return false
   end if

   # original 's' has at least 2 elements
```

```
    # pop 2nd value from ‘s’
    s2 = s.pop() # s:  [-> ??]

    # original ‘s’ has exactly 2 elements
    #   if ‘s’ is currently empty
    retval = s.empty()

    # push ‘s2’ back where it was
    s.push(s2) # s:  [-> s2 -> s3 -> ??]

    # push ‘s1’ back where it was
    s.push(s1) # s:  [-> s1 -> s2 -> s3 -> ??]

    return retval
end function
```

An alternative version:

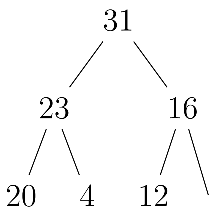
```
function is_double(s)
   return is_exactly(s, 2)
end function

function is_exactly(s, n)
   # Error checking
   if s == null or n < 0 then
      return false # or throw error
   end if

   n == 0 case
   if s.empty() then
      return n == 0
   end if

   # n >= 1 case
   # prepare an empty stack
   t = new stack()
   t.stack_init()

   retval = true

   # pop up to ‘n’ items out from ‘s’
   i = 0 # need to use the value of ‘i’ later
   for i < n; i++ do
      val = s.pop()
      t.push(val)

      # make sure we don’t dereference null pointers
      if s.empty() then
         retval = false
         break
```

```
        end if
    end for

    # If ‘retval’ is not set to false
    #   there are at least ‘n’ items in original ‘s’
    if retval then
        retval = s.empty() # true if exactly ‘n’ items
    end if

    # put the ‘i’ items we took out back into ‘s’
    for j = 0; j < i; j++ do
        val = t.pop()
        s.push(val)
    end for

    return retval
end function
```

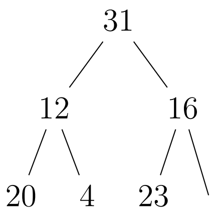
3. (a) Writer's note: this one can be copied from lecture's note. Be careful of the zero-based index.

```
function RecursiveSiftUp(A, n)
   # if the node value is larger than the parent
   if n > 0 and val > A[(n-1)//2] then
      # swap(A[n], A[(n-1)//2])
      val = A[n]
      A[n] = A[(n-1)//2]
      A[(n-1)//2] = val

      # check again
      RecursiveSiftUp(A, (n-1)//2)
   end if
   # otherwise do nothing and return
end function
```

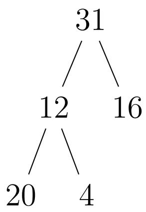
(b)



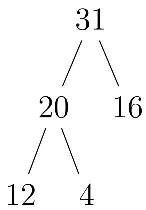
Swap node to be deleted to the last element



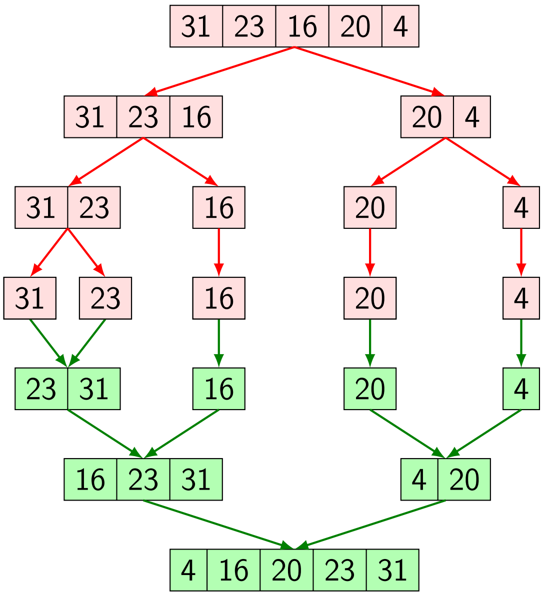
Dereference the last element



node 12 is smaller than its parent. Siftdown



(c)



(d) The complexity of a single mergesort for an array of size k is $\mathcal{O}(k \lg k)$. Merging two arrays of sizes k and l has a complexity of $\mathcal{O}(k + l)$.
Approach 1 gives a complexity of

$$\begin{aligned} C_1 &= \mathcal{O}(m \lg m + n \lg n + m + n) && (14) \\ &= \mathcal{O}(m \lg m + n \lg n) && (15) \\ &= \mathcal{O}(p \lg p), \quad p = \max(m, n) && (16) \end{aligned}$$

Approach 2 gives a complexity of

$$\begin{aligned} C_2 &= \mathcal{O}((m + n) \lg(m + n)) && (17) \\ &= \mathcal{O}(q \lg q), \quad q = m + n && (18) \end{aligned}$$

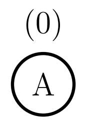
Since $m + n \geq \max(m, n)$, $C_2 \geq C_1$ and approach 2 has higher time complexity.

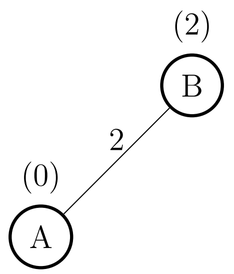
4. (a)

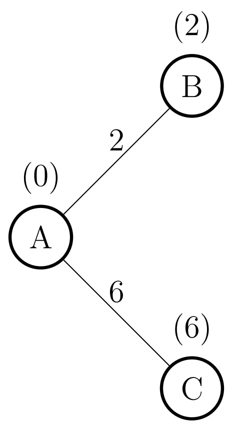
```
function DFS(adj, N)
   seq = new array(N)
   isVisited = new array(N)
   for i = 1; i <= N; i++ do
      isVisited[i] = false
   end for

   j = 1 # visit order
   for i = 1; i <= N; i++ do
      if not isVisited[i] then
         j = recursiveVisit(i, j, adj, isVisited, seq)
      end if

      if j == N then
         # not necessary but more efficient to cut it
         break
      end if
   end for
   return seq
end function
```

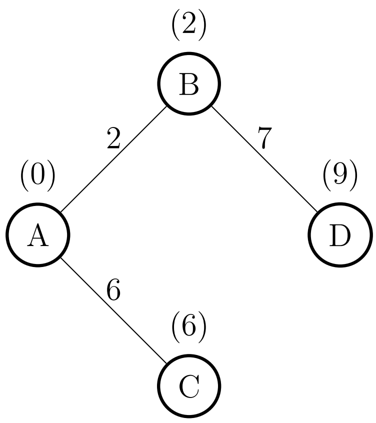
```
function recursiveVisit(i, j, adj, isVisited, seq)
   isVisited[i] = true
   seq[j] = i
   currAdjNode = adj[i]
   j++
   while currAdjNode != null do
      i = currAdjNode.data
      if not isVisited[i] then
         j = recursiveVisit(i, j, adj, isVisited, seq)
      end if
      currAdjNode = currAdjNode.next
   end while
   return j
end function
```

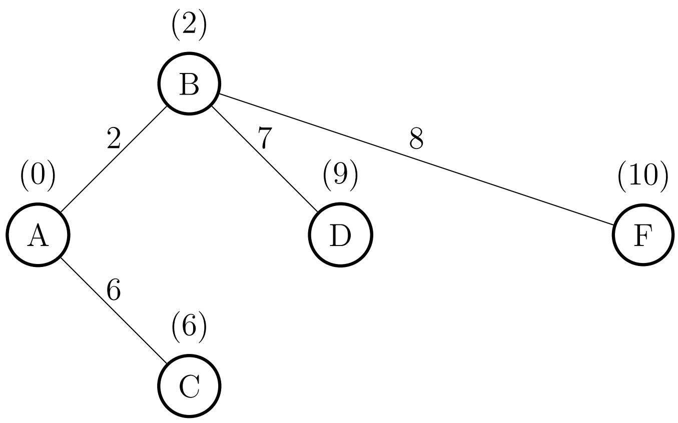
(b)

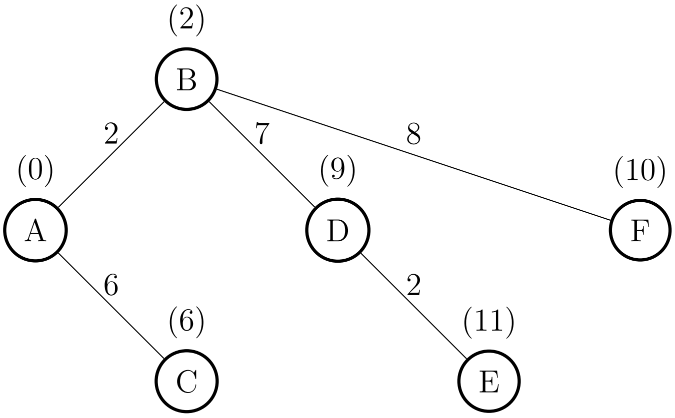







(2)
B
2
7
(0)
(9)
A
D
6
(6)
C


(2)
B
2
7
8
(0)
(9)
(10)
A
D
F
6
(6)
C



Summary:

Node	Shortest path from A	Minimum Cost
B	A $\rightarrow$ B	2
C	A $\rightarrow$ C	6
D	A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ C	9
E	A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ E	11
F	A $\rightarrow$ B $\rightarrow$ F	10

(c)

```
function AdjMatrixToLists(A, N)
   adjLists = new Array(N)
   for i = 1; i <= N; i++ do
      adjLists[i] = new LinkedListHead()
      node = adjLists[i]
      node.next = null
      for j = 1; j <= N; j++ do
         if i == j then
            continue
         end if
         if A[i, j] then
            node.next = new LinkedListNode()
            node.next.data = j
            node.next.next = null
            node = node.next
         end if
      end for
   end for
   return adjLists
end function
```

End of Paper