**NANYANG TECHNOLOGICAL UNIVERSITY**

**SEMESTER 1 EXAMINATION 2019-2020**

**EE2008 / IM1001 – DATA STRUCTURES AND ALGORITHMS**

November / December 2019

Time Allowed: 2½ hours

**INSTRUCTIONS**

1. This paper contains 4 questions and comprises 4 pages.
2. Answer all 4 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) Determine the asymptotic upper bound for the number of times the statement "$r = r + 1$" is executed in each of the following algorithms.

(i)
```
for i = 1 to n - 2
    for j = i + 2 to n
        r = r + 1
```

(ii)
```
i = n
while ( i > 1 ) {
    for j = 1 to i {
        r = r + 1
    }
    i = i - 2
}
```

(9 Marks)

(b) (i) Write a recursive algorithm to compute $n!$, where $n$ is a positive integer.

(ii) Set up the recurrence relation for the number of multiplications made by the algorithm in part (i) and solve it.

(8 Marks)

Note: Question No. 1 continues on page 2.



26

(c) Determine whether the following statements are true or false. If the statement is true, prove it. If the statement is false, give a counterexample.

(i) $\sum_{i=1}^{n} i \lg i = O(n^2 \lg n)$

(ii) $\sum_{i=0}^{k} \lg\left(\frac{n}{2^i}\right) = O(\lg^2 n)$ where $n = 2^k$.

(8 Marks)

2. (a) (i) A queue $Q$ is implemented using an array. Write an algorithm in pseudocode to determine whether $Q$ contains exactly one element. The algorithm should return *true* if $Q$ contains exactly one element. Otherwise, the algorithm should return *false*.

(ii) Draw the 11-item hash table resulting from hashing the keys 13, 23, 32, 55, 66, 89, 96 using the hash function $h(x) = x \bmod 11$. Assume that collisions are handled by **double hashing** using a second hash function $d(x) = 7 - (x \bmod 7)$.

(10 Marks)

(b) Using pseudocode, describe the implementation of the method *remove(p)* that removes the element at position *p* in the LIST ADT, assuming that the LIST ADT is implemented using a doubly linked list.

(5 Marks)

(c) Given a non-null pointer $T$ to the root of a binary search tree and a node $v$ in the tree, write a recursive algorithm in pseudocode to identify the node along the path from the node $v$ to the root that has the largest value.

(10 Marks)

3. (a) Heapify the following array to make it into a maxheap. Show and explain clearly what the array looks like in each step.

| | | | | | | |
|---|---|---|---|---|---|---|
| 34 | 45 | 12 | 52 | 38 | 23 | 19 |

(10 Marks)

Note: Question No. 3 continues on page 3.

(b) Continuing from your answer in part (a), show and explain clearly what the array looks like in each step when the 5th element in the heapified array is replaced by the value 60 and you want to maintain the array as a maxheap.

(5 Marks)

(c) Show each step clearly when radix sort is performed on the following list:

7349, 7211, 4456, 4324, 4208, 7212

(6 Marks)

(d) Is heapsort a stable sorting algorithm? If yes, prove it, otherwise provide a counterexample to justify your answer.

(4 Marks)

4. (a) Consider the weighted graph whose adjacency list is shown in Figure 1. Use Dijkstra's algorithm to find the shortest path from vertex 2 to vertex 5. Show each step clearly.

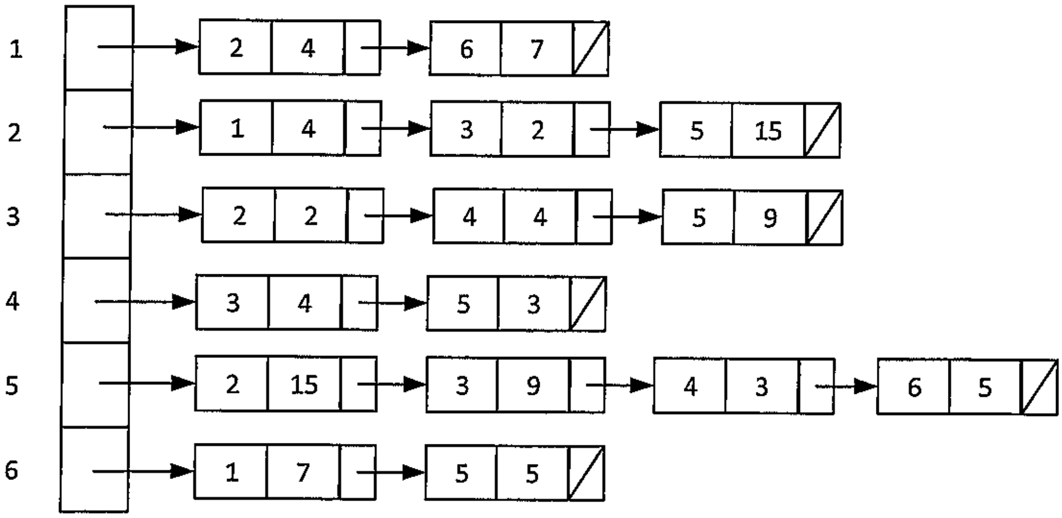



**Figure 1**

(12 Marks)

Note: Question No. 4 continues on page 4.



27

(b) Use Prim's algorithm starting at vertex 2 to find a minimum spanning tree in the weighted graph shown in Figure 1. Show each step clearly.

(8 Marks)

(c) Can depth-first search be used to find the shortest path from a vertex to another vertex in an unweighted graph? Justify your answer.

(5 Marks)

END OF PAPER

1a (i)

for $i = 1$ to $n-2$
  for $j = i+2$ to $n$
    $r = r+1$

Rewrite the number of times the statement is executed in sigma ($\Sigma$) form.

$$\text{Number of times} = \sum_{i=1}^{n-2}\sum_{j=i+2}^{n} 1 \quad \rightarrow \text{The statement is run 1 time for each loop.}$$

$$= \sum_{i=1}^{n-2} n-(i+2)+1 \quad \rightarrow \text{There are } n-(i+2)+1 \text{ counts.}$$

$$= \sum_{i=1}^{n-2} n-i-1$$

$$= (n-2)+(n-3)+(n-4)+\dots+\underbrace{(n-(n-3)-1)}_{=2}+\underbrace{(n-(n-2)-1)}_{=1}$$

$$\leq \underbrace{n+n+n+\dots+n}_{(n-2)\text{ times}} = n\cdot(n-2)$$

$$\leq n\cdot n$$

$$= O(n^2)$$

(ii)

$i = n$
while ( $i > 1$ ) {
  for $j = 1$ to $i$ {
    $r = r+1$
  }
  $i = i-2$
}

The outer while loop runs from $i=n$ down to when $i$ is just larger than 1.
Assume the smallest $i$ is 2, the while statement is equivalent to:

$$\sum_{\substack{i=2 \\ i=i+2}}^{n} \dots \quad \rightarrow \text{The number of times this is executed is roughly } \frac{n}{2} \text{ times.}$$

↳ This is because the interval is 2 instead of 1.

$$\text{Number of times} = \sum_{\substack{i=2 \\ i=i+2}}^{n}\sum_{j=1}^{i} 1 = \sum_{\substack{i=2 \\ i=i+2}}^{n} i$$

$$= 2+4+6+\dots+n-2+n$$

$$\leq \underbrace{n+n+n+\dots+n}_{n/2\text{ times}} = n\cdot\frac{n}{2}$$

$$= \frac{1}{2}n^2$$

$$= O(n^2)$$

> This type of question is usually easy to score!
> The number of nested loops you see is usually (not necessarily always!) the power of n.
> e.g. if you see:
> for i = 1 to n
>   for j = i to n
>     for k = j to n
>       r = r + 1
>
> The upper bound is likely to be $O(n^3)$.
> It will never be lower power such as $O(n^2)$!

1b (i) $n! = n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot 3\cdot 2\cdot 1$

Base case is when $n = 1$ (since the question states that $n > 0$).

$\boxed{\text{Notice that } n! = n\cdot(n-1)!}$ $\rightarrow$ Multiply recursively.

28

```
factorial (n) :
    if n == 1 :              → Base case.
        return 1
    return n * factorial (n-1)   → Recursive call. From the property n! = n·(n-1)!
```

(ii) Recurrence relations are also stated recursively.

$$
\begin{aligned}
T(n) &= 1 + T(n-1) && \rightarrow \text{Main operation is multiplication.} \\
&= 1 + (1 + T(n-1-1)) && \text{Perform multiplication 1 times / call.} \\
&= 2 + T(n-2) \\
&= 3 + T(n-3) = \ldots = n-1 + \underbrace{T(n-(n-1))}_{= T(1) = 0} \\
&\leq n = \underline{\underline{O(n)}}.
\end{aligned}
$$

1c (i)

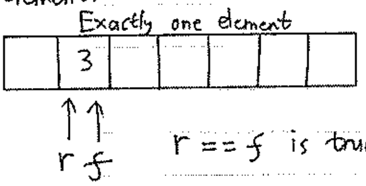
$$
\begin{aligned}
\sum_{i=1}^{n} i \lg i &= 1\lg 1 + 2\lg 2 + 3\lg 3 + \ldots + n\lg n \\
&\leq \underbrace{n\lg n + n\lg n + n\lg n + \ldots + n\lg n}_{n \text{ times}} \\
&= (n\lg n)(n) \\
&= O(n^2 \lg n) \qquad \therefore \text{The statement is true.}
\end{aligned}
$$

(ii)

$$
\begin{aligned}
\sum_{i=0}^{k} \lg\left(\frac{n}{2^i}\right) &= \lg\left(\frac{n}{2^0}\right) + \lg\left(\frac{n}{2^1}\right) + \lg\left(\frac{n}{2^2}\right) + \ldots + \lg\left(\frac{n}{2^k}\right) \qquad ; \; n = 2^k \\
&= \lg(2^k) + \lg(2^{k-1}) + \lg(2^{k-2}) + \ldots + \lg(2^1) + \lg(2^0) \\
&\leq \underbrace{\lg(2^k) + \lg(2^k) + \lg(2^k) + \ldots + \lg(2^k)}_{(k+1) \text{ times}} \\
&= (k+1)\cdot \underbrace{\lg 2^k}_{\log_2 2^k = k} = (k+1)\cdot k = k^2 + k \\
&= \lg^2 n + \lg n \\
&= \underline{\underline{O(\lg^2 n)}} \qquad \therefore \text{The statement is true.}
\end{aligned}
$$

$n = 2^k \downarrow$ $k = \log_2 n$, $\underline{k = \lg n}$

2a (i) For a queue, the pointer r and f point to the same index if there is only one element.




r == f is true.

But do not forget to exclude the case where the queue is empty, in which r = f as well (r = f = -1).

```
one_element () :
    if (r == -1) :     // Check if Q empty.
        return 0       // False
    if (r == f) :
        return 1
    else return 0
```

Alternative (more concise):

```
one_element ():
    if (r == -1):
        return 0
    return r == f
```

The expression r == f evaluates to True if r = f, false if r ≠ f.

(ii) $h(x) = x \bmod 11$
$d(x) = 7 - (x \bmod 7)$ } Probe $= (h(x) + j\,d(x)) \bmod 11$

| Key | h(x) | d(x) | Probes |
|---|---|---|---|
| 13 | 2 | 7−6 = 1 | 2 |
| 23 | 1 | 7−2 = 5 | 1 |
| 32 | 10 | 7−4 = 3 | 10 |
| 55 | 0 | 7−6 = 1 | 0 |
| 66 | 0 | 7−3 = 4 | 0 \| 4 |
| 89 | 1 | 7−5 = 2 | 1 \| 3 |
| 96 | 8 | 7−5 = 2 | 8 |

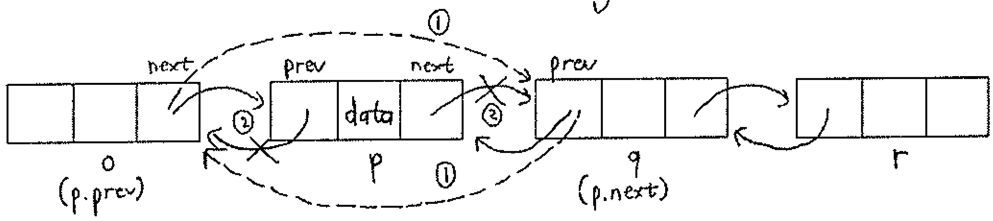
→ Already used.

Firstly compute h(k) and d(k).
Then, the probes will be given by

$(h(k) + j\,d(k)) \bmod 11$

$j = 0, 1, 2, \ldots$
→ Start at $j = 0$.
→ If colision still hapens, increase $j$ by 1.

2b

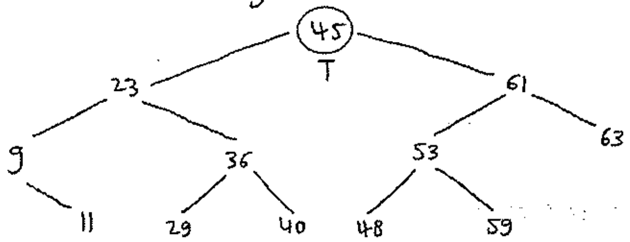



To remove the element p, we:
① Reasign the pointers of neighbouring elements so p is entirely skipped.
② Remove the pointers from p so that it links to no other element.
Doing this renders p inaccessible and isolated, effectively removing it from the list.

```
remove (p):                    // p is a pointer to the element p.
    data = p.data              // store the data inside p to return it later.
    if p.next != null:         // != is not equal to.
        p.next.prev = p.prev   // step ①. Account for the fact that p might be the first/last
    if p.prev != null:         //    element.
        p.prev.next = p.next
    p.next = null              // step ②
    p.prev = null
    return data                // Finally, return data inside p.
```

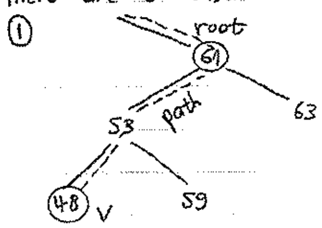
2c Task: to identify the element with the largest value along the path from v to the root T. (note we can perform the search the other way around ; from T to v).

Recall the special property of a binary search tree ; for a given node, all elements smaller than that node is placed on the left subtee, all elements larger than that node on the right subtree.
Consider the following BST:



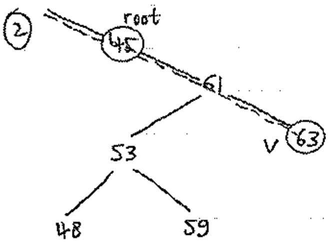

Property of BST
→ Go left: get a smaller value.
→ Go right: get a larger value.
We would like to find the node with the maximum value. Thus, there is no need to go left from a node on the path as it will give us a smaller value (we need larger!).

There are 3 cases to consider...

① 


If v is smaller than the current root T, the path from T to v will take us to the left subtree, inside which all nodes are smaller than T. Thus, the maximum value along the path is T.

→ Stop the search, return T.data.

② 

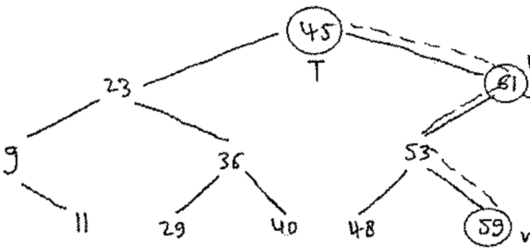

If v is larger than the current root T, the path takes us to the right subtree (there are still larger value). Set the right child as the new root and repeat the check.

Recursive call.

③ If the root T is equal to v, then we must have gone all the way right (if we went left then the algorithm must have finished earlier). This means v is actually the maximum

Assume that the root node T and the node v are also considered (i.e. if they turn out to be the largest then return their value).

```
max_path (T, v) :
    if v < T.data :          // Case ①
        return T.data
    if v > T.data :          // Case ②
        return max_path (T.right, v)
    else :                   // Case ③
        return v
```

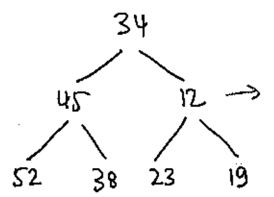



Maximum! Once we go left, subsequent values are all smaller than 61!

3a

| 34 | 45 | 12 | 52 | 38 | 23 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

↑




→ Start siftdown here.

Heapify by applying siftdown from the index n/2 (7/2 = 3) down to n = 1.

**Indexing of maxheap:**
left_child = 2* parent
right_child = 2* parent + 1
parent = child / 2 (floor division)

siftdown (3)

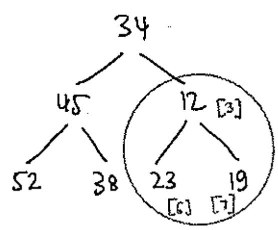
| 34 | 45 | 12 | 52 | 38 | 23 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

swap

| 34 | 45 | 23 | 52 | 38 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

**Heapify**

Element 3 is smaller than both children! Swap with largest child.



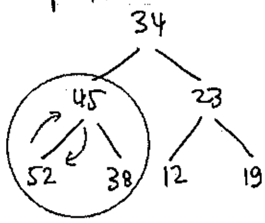
| 34 | 45 | 23 | 52 | 38 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

siftdown (2)

| 34 | 45 | 23 | 52 | 38 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

swap

Swap indices 2 and 4.




| 34 | 52 | 23 | 45 | 38 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

siftdown (1)

| 34 | 52 | 23 | 45 | 38 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

swap

Swap 1 and 2.

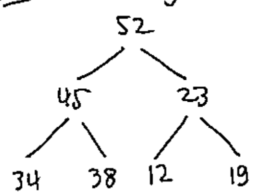
| 52 | 34 | 23 | 45 | 38 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

swap

Since index 2 still has children, don't forget to compare index 2 with its children too (siftdown (2) again).
Swap index 2 and 4.

| 52 | 45 | 23 | 34 | 38 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Finish. Final array is this one as can be confirmed by drawing the maxheap.




3b

| 52 | 45 | 23 | 34 | 60 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

To maintain maxheap structure, compare the $5^{th}$ element with its children & parent, perform siftdown / siftup if necessary. (index 5 has no children)

| 52 | 45 | 23 | 34 | 60 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

60 > 45. Perform siftup.

| 52 | 60 | 23 | 34 | 45 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Check again with parent.
Swap index 1 and 2.

| 60 | 52 | 23 | 34 | 45 | 12 | 19 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Finish.

3c Perform radix sort → sort by least significant digit up to most significant digit.

When performing radix sort, do not change the order of 2 items that are identical, e.g. when sorting 4, 3, <u>6</u>, 9, 7, <u>6</u> you should come up with 3, 4, <u>6</u>, <u>6</u>, 7, 9 instead of 3, 4, <u>6</u>, <u>6</u>, 7, 9. This will make a difference later!

30

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7349 | | 7211 | | 4208 | | 4208 | | 4208 |
| 7211 | | 7212 | | 7211 | | 7211 | | 4324 |
| 4456 | $\longrightarrow$ | 4324 | $\longrightarrow$ | 7212 | $\longrightarrow$ | 7212 | $\longrightarrow$ | 4456 |
| 4324 | Sort by ones. | 4456 | Sort by tens | 4324 | Sort by hundreds. | 4324 | Sort by thousands. | 7211 |
| 4208 | | 4208 | | 7349 | | 7349 | | 7212 |
| 7212 | | 7349 | | 4456 | | 4456 | | 7349 |
| | | ↓ Sorted ascending ones. | | ↓ Sorted ascending tens. | | ↓ Sorted ascending hundreds | | Finish. |

3d For a sorting algorithm to be stable, the order of elements with the same value don't change, e.g. 4, 3, $\underline{6}$, 9, 7, $\underline{\underline{6}}$ is sorted into 3, 4, $\underline{6}$, $\underline{\underline{6}}$, 7, 9 instead of 3, 4, $\underline{\underline{6}}$, $\underline{6}$, 7, 9.
Heapsort involves <u>swapping elements in the array</u>, which means that it is likely to be <u>not stable</u>. To verify our initial guess, use a numeric example.

Consider the following array where we have 2 numbers of the same value.

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| $\underline{7}$ | 4 | 2 | $\underline{\underline{7}}$ |

To sort, we start by heapifying the array.

siftdown (2)

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| $\underline{7}$ | $\underline{\underline{7}}$ | 2 | 4 |

siftdown (1)

$\longrightarrow$

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| $\underline{7}$ | $\underline{\underline{7}}$ | 2 | 4 |

(heapified)

After heapifying, <u>swap A[1] with A[4].</u>

| 1 | 2 | 3 | ↑ |
|---|---|---|---|
| 4 | $\underline{\underline{7}}$ | 2 | $\underline{7}$ |

Treat this as a new maxheap.

Then, proceed by heapifying A[1 : 3].

| 1 | 2 | 3 | |
|---|---|---|---|
| $\underline{\underline{7}}$ | 4 | 2 | $\underline{7}$ |

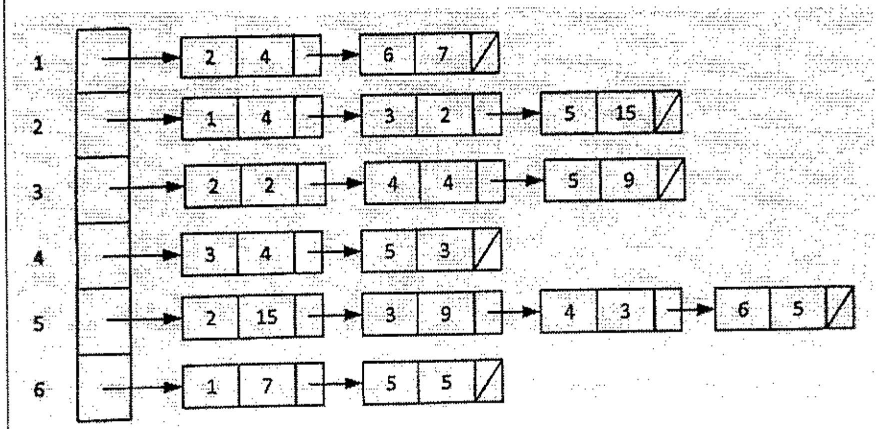
... and swap A[1] with A[3].

| 1 | 2 ↑ | | |
|---|---|---|---|
| 2 | 4 | $\underline{\underline{7}}$ | $\underline{7}$ |

Treat as maxheap

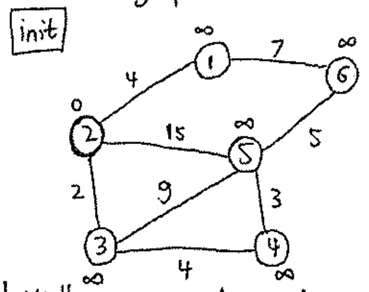
Continuing this process, we can see that we will obtain 2, 4, $\underline{\underline{7}}$, $\underline{7}$. However, $\underline{7}$ comes before $\underline{\underline{7}}$ in the original array! Thus, <u>heapsort is not stable.</u>

4a

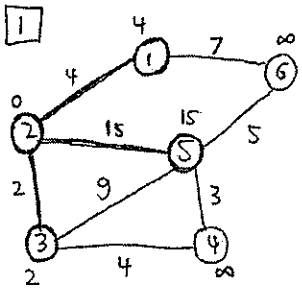



→ Start from vertex 2.
→ All other vertices have initial weight of ∞.
→ Update the weight of other vertices ("relaxing" the weights) and include the reached vertices in the SPT (shortest path tree).
→ Choose vertex with smallest weight, repeat.

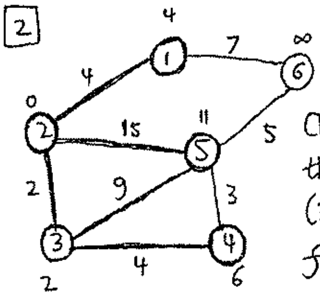
Sketch the graph so it is easier to perform the algorithm.



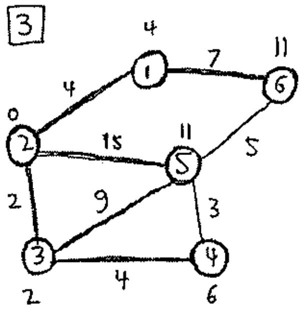
Initially start at vertex 2.



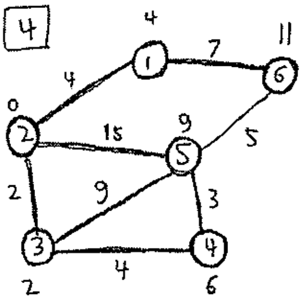
Relax all vertices adjacent to 2 (1,3,5), update weights if weights become smaller, and include them in the SPT.



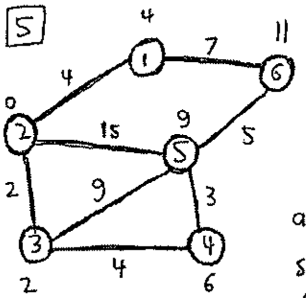
Choose vertex with the smallest weight (3), then relax from that vertex.



Relax from vertex 1 (smallest weight and haven't been used).



Relax from vertex 4.



Relax from vertex 6. Vertex 5 is actually our destination so do not relax from vertex 5!

We stop here since all vertices have been added to SPT. The final weight of vertex 5 from vertex 2 is 9 → shortest path = 9.

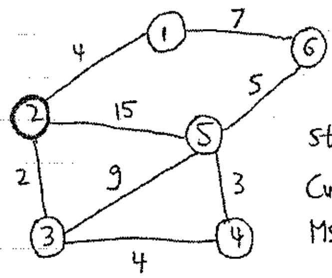
| Current Vertex | SPT | Weights relative to vertex 2 | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | |
| 2 | 2 | ∞ | 0 | ∞ | ∞ | ∞ | ∞ | init |
| 2 | 2, 1, 3, 5 | 4 | 0 | 2 | ∞ | 15 | ∞ | 1 |
| 3 | 2, 1, 3, 5, 4 | 4 | 0 | 2 | 6 | 11 | ∞ | 2 |
| 1 | 2, 1, 3, 5, 4, 6 | 4 | 0 | 2 | 6 | 11 | 11 | 3 |
| 4 | 2, 1, 3, 5, 4, 6 | 4 | 0 | 2 | 6 | 9 | 11 | 4 |
| 6 | 2, 1, 3, 5, 4, 6 | 4 | 0 | 2 | 6 | 9 | 11 | 5 |
| — | — | 4 | 0 | 2 | 6 | 9 | 11 | finish |

## Tips :

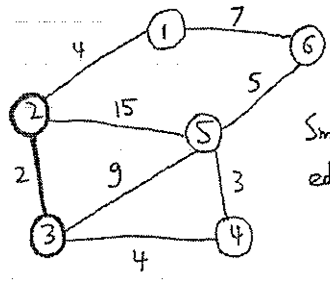
On answer book you can just sketch the graph once and put your result in a table form (see left). You can do the step by step tracing (like I did above) on your scratch paper to help you fill in the actual table on the answer book.

4b For prim's algorith, we'll continuously add edges to our MST set until the number of edges we have = number of vertices − 1.
The edge we add is the one with the smallest weight.

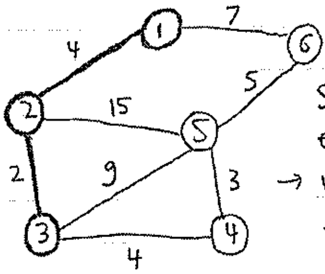
Sketch the graph:



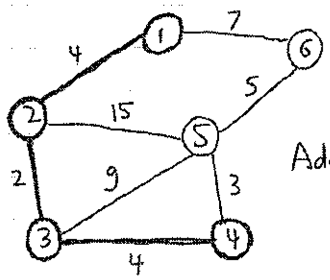
Start from vertex 2.
Currently no edges in the MST set.



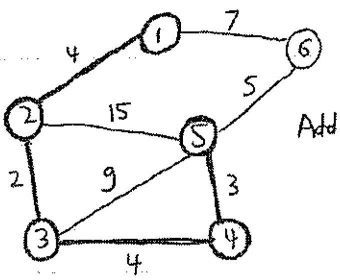
Smallest weight: edge (2,3).



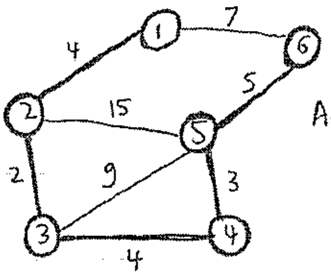
Smallest weight: edge (1,2).
→ We do not pick the edge (4,5) although weight is smaller since that edge is adjacent to our MST.



Add edge (3,4).



Add edge (4,5).

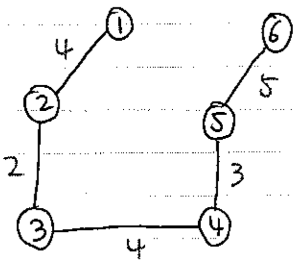


Add edge (5,6).

We already have 6 − 1 = 5 edges inside our MST set, stop algorithm.

| No | Vertices in MST | MST Set |
|---|---|---|
| – | 2 | – |
| 1 | 2,3 | (2,3) |
| 2 | 2,3,1 | (2,3), (1,2) |
| 3 | 2,3,1,4 | (2,3), (1,2), (3,4) |
| 4 | 2,3,1,4,5 | (2,3), (1,2), (3,4), (4,5) |
| 5 | 2,3,1,4,5,6 | (2,3), (1,2), (3,4), (4,5), (5,6) |
| | | Total weight = 18 |

MST:

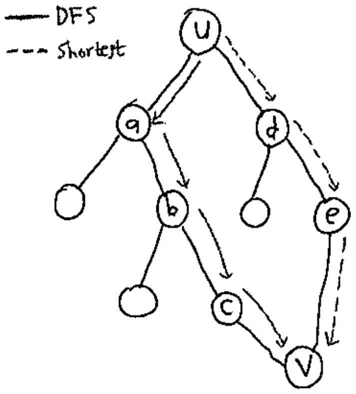


4C Depth First Search (DFS) works by accessing vertices as deep as possible before checking other nodes on the same depth.

(next page)

While DFS is great to search for a vertex in a given graph, it cannot be used to find shortest path in an unweighted graph since the algorithm immediately stops once it finds the searched vertex, without checking if there is actually a shorter path.

Consider the following graph:




When DFS is run on this graph to find shortest path from vertices u to v, it will first find the path u, a, b, c, v (4 edges) and returns that path before it finds the other path u, d, e, v (3 edges), which is actually shorter.

∴ Hence DFS <u>cannot</u> be used to find shortest path.

Remember to do questions that you feel is the easiest for you first.
→ Qn 1 should be straightforward (complexity, recurrence relation, etc).

For writing algorithms...
→ Recursive algo: try to discover operations that is performed over and over again / operations breakable into smaller, same operation.
→ Do not forget special cases (e.g. Qn 2a (i): what if the queue is empty?)

All the Best :)