SC1007 Tutorial 5
Hash Table and Graph

Dr Liu Siyuan

Email: syliu@ntu.edu.sg

Office hour: Monday & Wednesday 4-5pm

Office: N4-2C-117a

Hash table (review)

What is hashing?

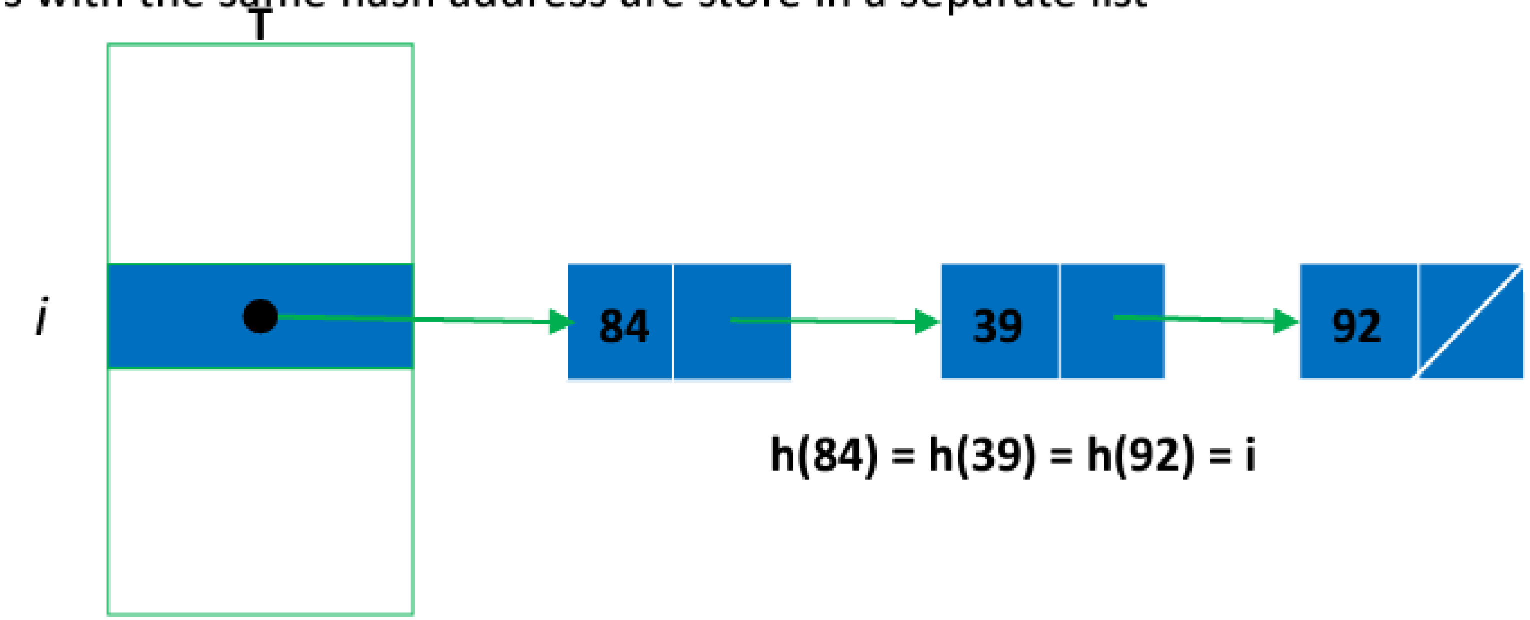
- To reduce the key space to a reasonable size
- Each key is mapped to a unique index (**hash value/code/address**)
- Search time remains O(1) on the average

hash function: {all possible keys} → {0, 1, 2, …, h-1}

- The array is called a **hash table**
- Each entry in the hash table is called a **hash slot**
- When multiple keys are mapped to the same hash value, a **collision** occurs
- If there are n records stored in a hash table with h slots, its **load factor** is $\boldsymbol{\alpha} = \frac{n}{h}$

Closed Addressing: Separate Chaining

- Keys are not stored in the table itself
- All the keys with the same hash address are store in a separate list

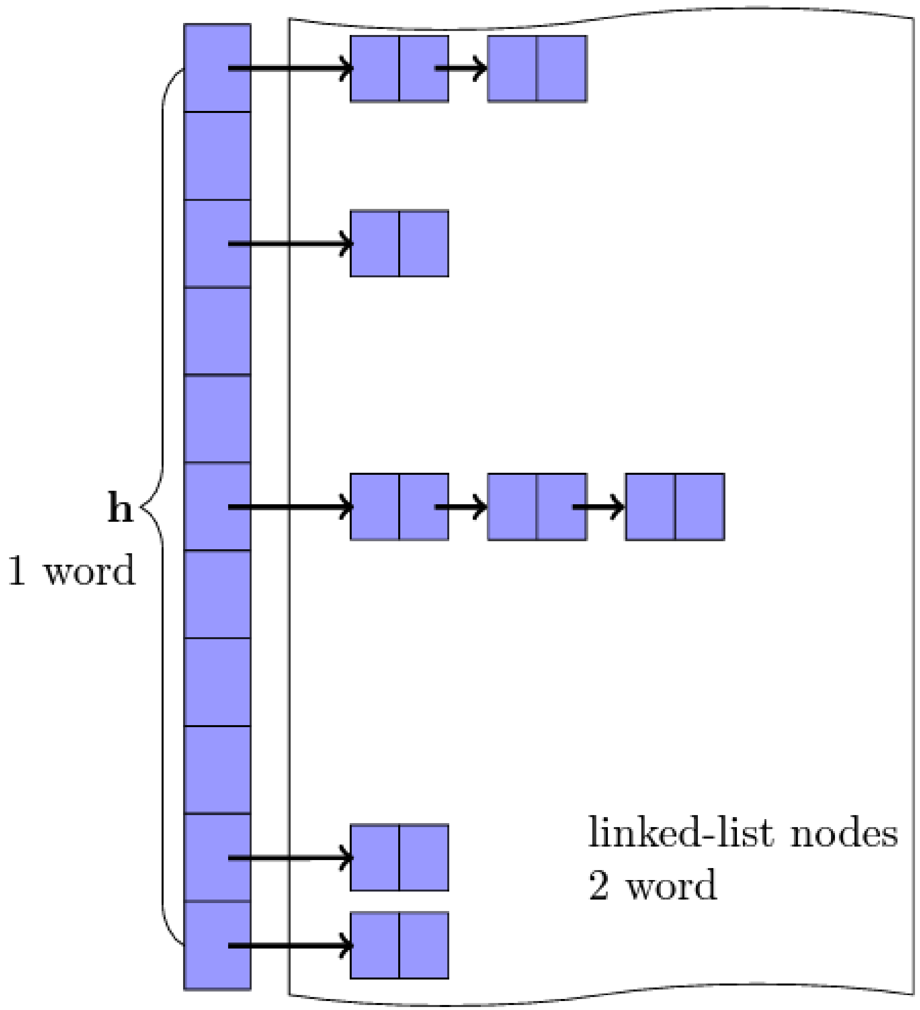


- During searching, the searched element with hash address i is compared with elements in linked list H[i] sequentially
- In closed address hashing, there will be α number of elements in each linked list on average.

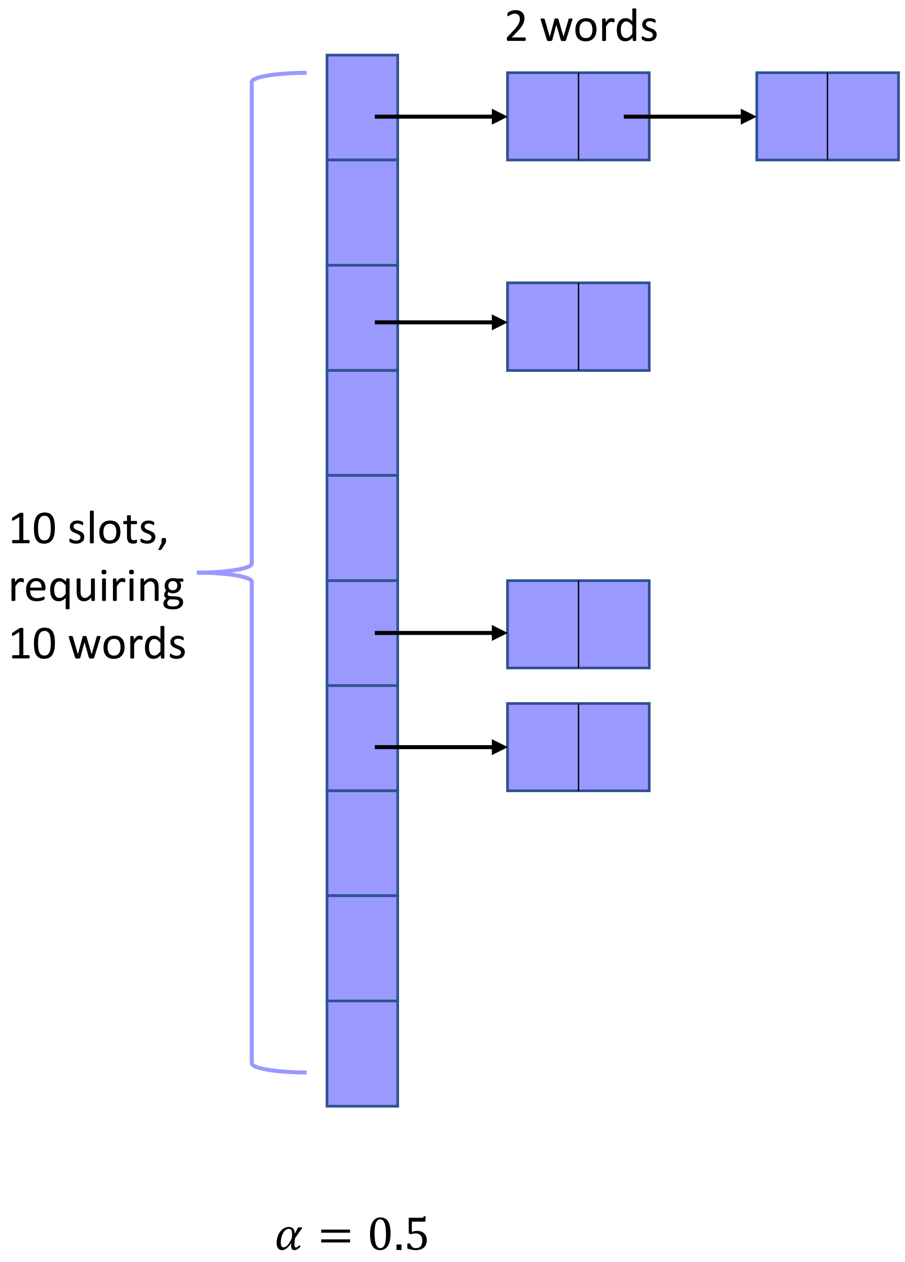
Open Addressing

- Keys are stored in the table itself
- When collision occurs, probe is required for the alternate slot
  - Linear Probing
    - $H(k,i) = (H(k)+i) \bmod h$, $i = 0, 1, 2, \ldots, h-1$
  - Quadratic Probing
    - $H(k,i) = (H(k)+c_1 i+c_2 i^2) \bmod h$, $i = 0, 1, 2, \ldots, h-1$
  - Double Hashing
    - $H(k,i) = (H(k)+iD(k)) \bmod h$, $i = 0, 1, 2, \ldots, h-1$

Q1

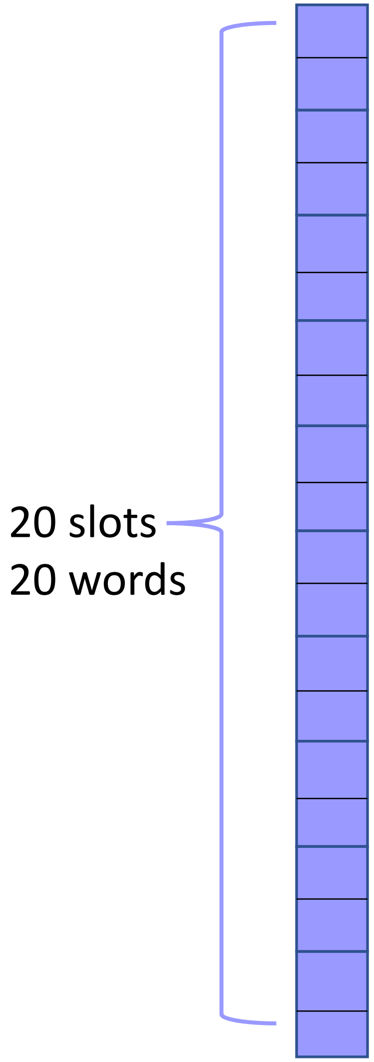


- The type of a hash table *H* under closed addressing is an array of list references, and under open addressing is an array of keys. Assume a key requires one "word" of memory and a linked list node requires two words, one for the key and one for a list reference.
- Consider each of these load factors for closed addressing: 0.5, 1.0, 2.0. Estimate the total space requirement, including space for lists, under closed addressing
- Assuming that the same amount of space is used for an open addressing hash table, what are the corresponding load factors under open addressing?



- Let *h* be hash table size. There are *h* slots.
- Load factor $\alpha = \frac{n}{h}$

1. When $\alpha = 0.5$, under closed addressing
   - *n*=0.5*h*, meaning there are 0.5*h* keys, which are 0.5*h* nodes.
   - Each node require 2 words.
   - Total space is $2\mathrm{n} + h = 2 \times 0.5h + h = 2h$.
2. When *α*=1
   - There are *h* nodes
   - Total space: $h \times 2 + h = 3h$.
3. When *α*=2
   - There are 2*h* nodes



- Assuming that the same amount of space is used for an open addressing hash table, what are the corresponding load factors under open addressing?

1. When there are 0.5*h* keys, and given 2*h* space, the corresponding load factor under open addressing is $\alpha = \frac{0.5h}{2h}$=0.25
2. When there are 1*h* keys, and given 3*h* space,
$$\alpha = \frac{h}{3h} = 0.33$$
3. When there are 2*h* keys, and given 5*h* space,
$$\alpha = \frac{2h}{5h} = 0.4$$

Q2

- Consider a hash table of size *n* using open address hashing and linear probing. Suppose that the hash table has a load factor of 0.5, describe with a diagram of the hash table, the best-case and the worst-case scenarios for the key distribution in the table.
- For each of the two scenarios, compute the average-case time complexity in terms of the number of key comparisons when inserting a new key. You may assume equal probability for the new key to be hashed into each of the *n* slots. [Note: Checking if a slot is empty is not a key comparison.]

n=7

$H(k)$ = k mod n

0	0
1	
2	2
3	
4	4
5	
6	6

- Linear Probing: probe the next slot when there is a collision
  - $H(k,i)$ = $(k+i)$ mod n, where $i \in [0, n-1]$
- There are n slots, α=0.5, there are $n/2$ keys.
- Best case scenario:
  - The n/2 keys are hashed and distributed evenly into the n slots
- Assume that equal probability for a key to be hashed into each of the n slots, the average-case time complexity

$$= \frac{1}{n}\left(\sum_{i=1}^{\frac{n}{2}} 0\right) + \frac{1}{n}\left(\sum_{i=1}^{\frac{n}{2}} 1\right) = \frac{1}{n} \times \frac{n}{2} = 0.5 = \Theta(1)$$

n=7

$H(k)$ = *k* mod n

0	
1	
2	
3	3
4	10
5	17
6	24

- Linear Probing: probe the next slot when there is a collision
    - $H(k,i)$ = $(k+i)$ mod n, where $i \in [0, n\text{-}1]$
- There are *n* slots, α=0.5, there are *n*/2 keys.
- Worse case scenario:
    - The n/2 keys are hashed in consecutive slots. Each key always has to rehash and visit every key in the table. The ith key is hashed and rehash i times to get the slot.
- Average-time-complexity

$$= \frac{1}{n}\left(\sum_{i=1}^{\frac{n}{2}} 0\right) + \frac{1}{n}\left(\sum_{i=1}^{\frac{n}{2}} i\right) = \frac{1}{n} \times \frac{\frac{n}{2}\times(1+\frac{n}{2})}{2}$$

$$= \frac{n}{8} + \frac{1}{4} = \Theta(n)$$

Q3

- Manually execute breadth-first search on the undirected graph in Figure 5.2, starting from vertex s. Then, use it as an example to illustrate the following properties:
- (a) The results of breadth-first search may depend on the order in which the neighbours of a given vertex are visited.
- (b) With different orders of visiting the neighbours, although the BFS tree may be different, the distance from starting vertex *s* to each vertex will be the same.

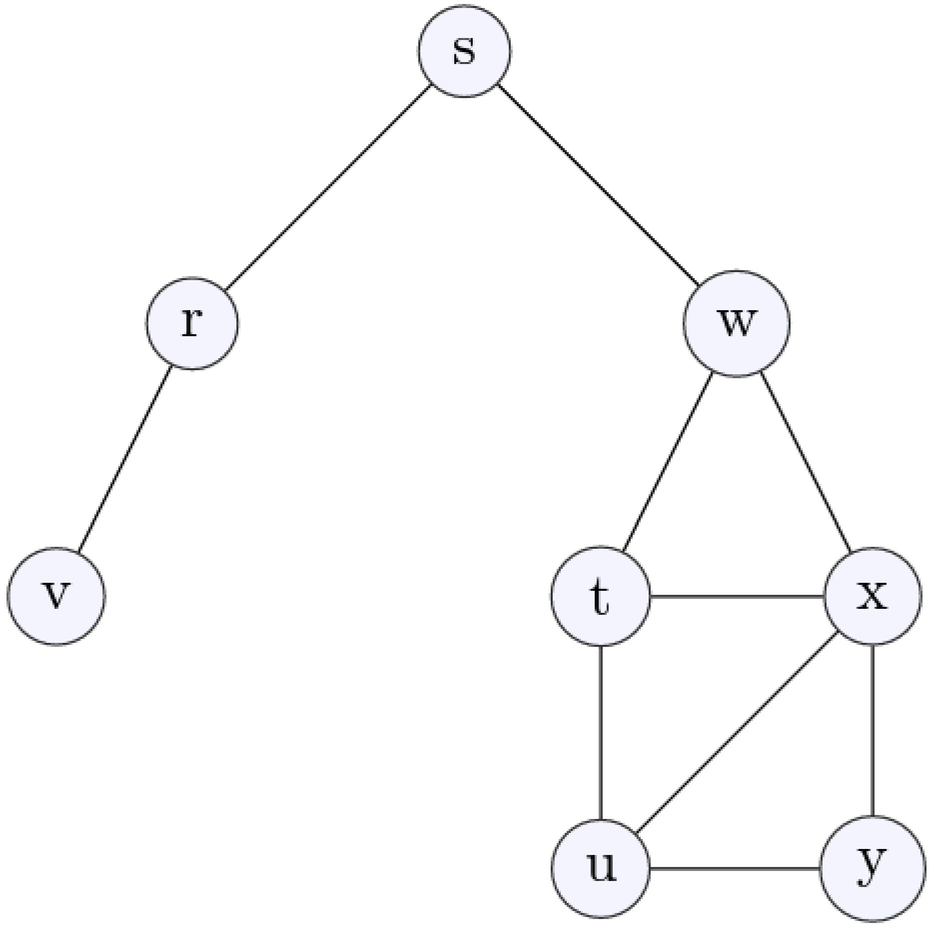


Figure 5.2: The Graph for Q3

Breadth first search (BFS) review

- Work similar to level order traversal of trees
- BFS systematically explores the edges directly connected to before visiting vertices further away.
- A queue is used to monitor which vertices to visit the next

BFS Algorithm

---

function BFS(Graph G, Vertex v)
  create a Queue, Q
  enqueue v into Q
  mark v as visited
  while Q is not empty **do**
    dequeue a vertex denoted as w
    for each unvisited vertex u adjacent to w **do**
      mark u as visited
      enqueue u into Q
    end for
  end while
end function

BFS Algorithm

```
function BFS(Graph G, Vertex v)
    create a Queue, Q
    enqueue v into Q
    mark v as visited
    while Q is not empty do
        dequeue a vertex denoted as w
        for each unvisited vertex u adjacent to w do
            mark u as visited
            enqueue u into Q
        end for
    end while
end function
```

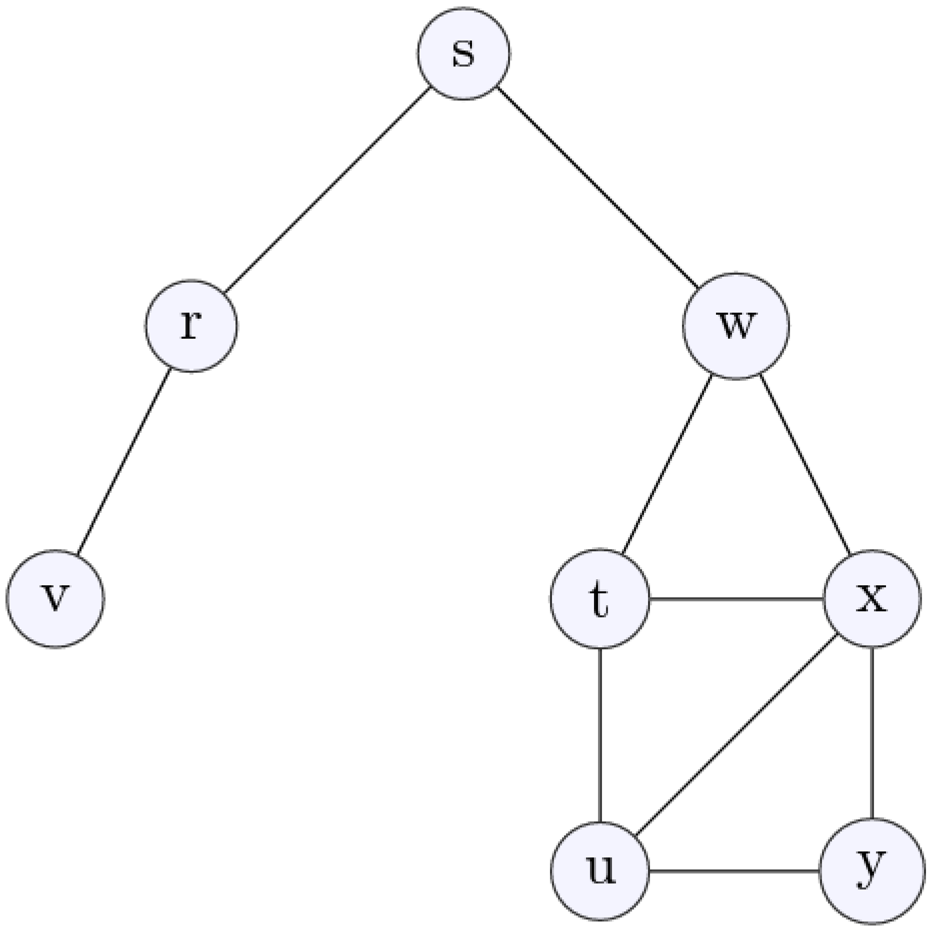


Figure 5.2: The Graph for Q3

Visiting neighbours in alphabetical order

The sequences visited: s r w v t x u y

The queue Q: s r w v t x u y

BFS Algorithm

```
function BFS(Graph G, Vertex v)
    create a Queue, Q
    enqueue v into Q
    mark v as visited
    while Q is not empty do
        dequeue a vertex denoted as w
        for each unvisited vertex u adjacent to w do
            mark u as visited
            enqueue u into Q
        end for
    end while
end function
```

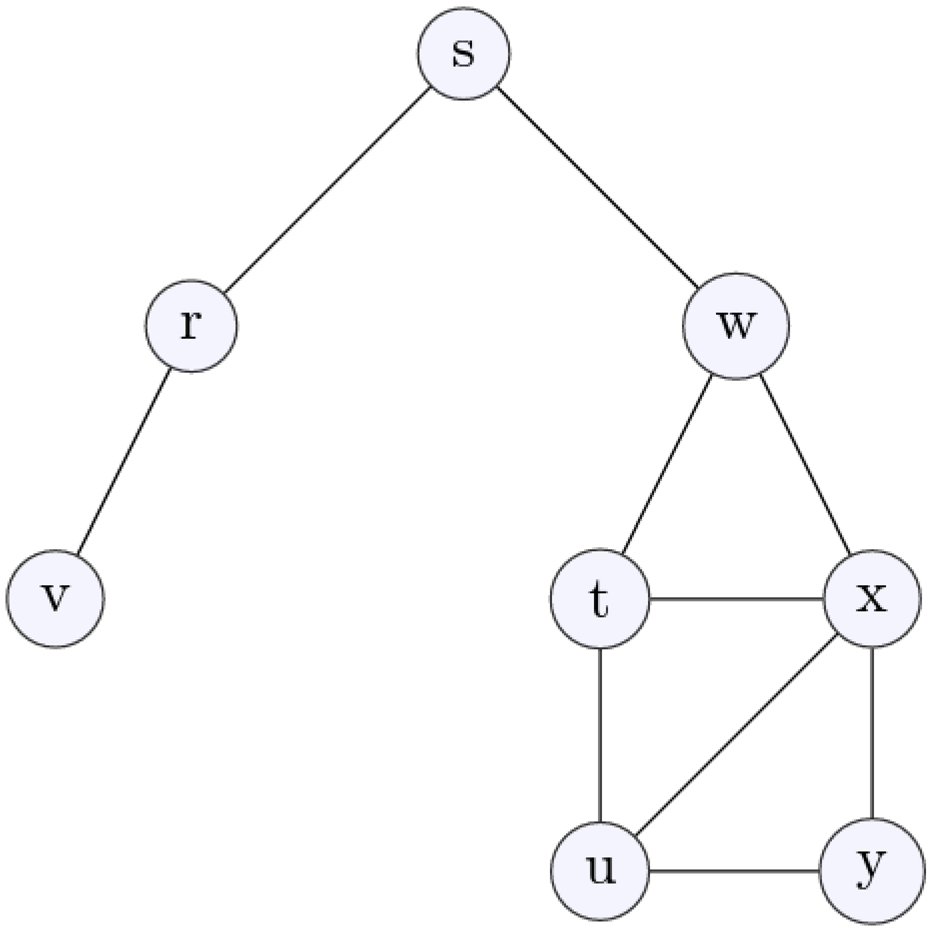


Figure 5.2: The Graph for Q3

Visiting neighbours in reverse alphabetical order

The sequences visited: s w r x t v y u

The queue Q: s w r x t v y u

- Let G be a graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G

BFS Algorithm

---

function BFS(Graph G, Vertex v)
  create a Queue, Q
  enqueue v into Q
  mark v as visited
  while Q is not empty **do**
    dequeue a vertex denoted as w
    for each unvisited vertex u adjacent to w **do**
      mark u as visited add edge (w, u)
      enqueue u into Q
    end for
  end while
end function

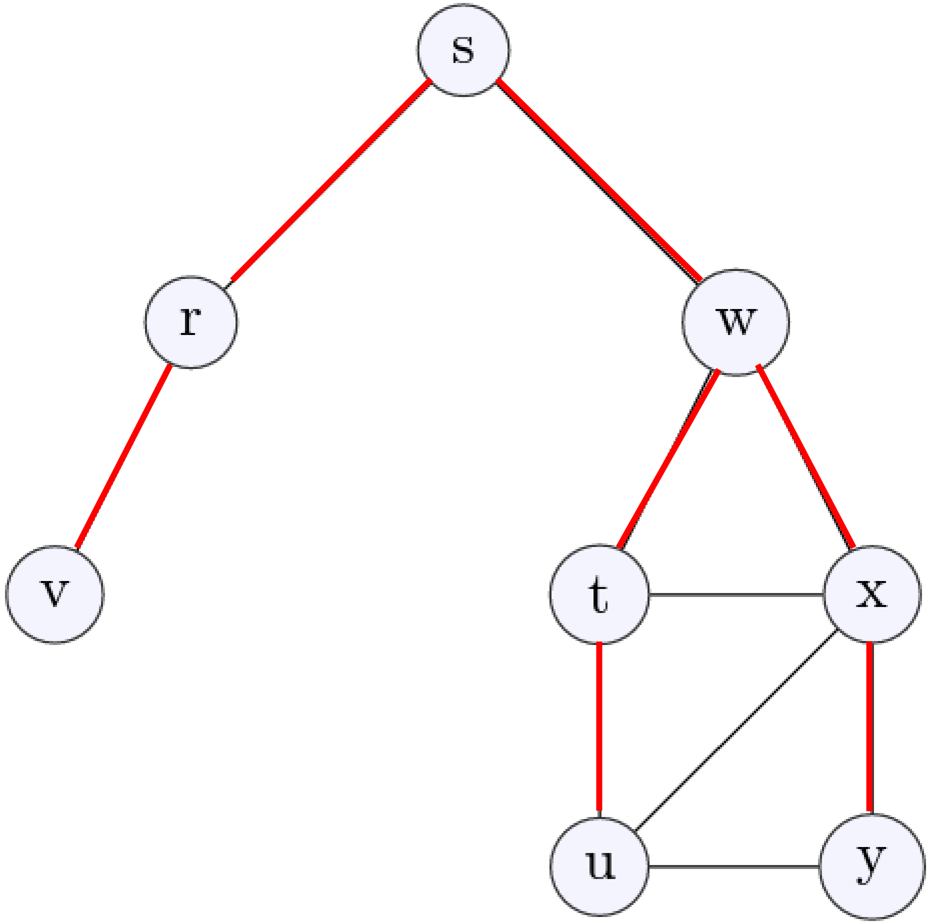


Figure 5.2: The Graph for Q3

Visiting neighbours in alphabetical order
The sequences visited: s r w v t x u y
The queue Q: s r w v t x u y

- Let G be a graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G

BFS Algorithm

```
function BFS(Graph G, Vertex v)
    create a Queue, Q
    enqueue v into Q
    mark v as visited
    while Q is not empty do
        dequeue a vertex denoted as w
        for each unvisited vertex u adjacent to w do
            mark u as visited    add edge (w, u)
            enqueue u into Q
        end for
    end while
end function
```

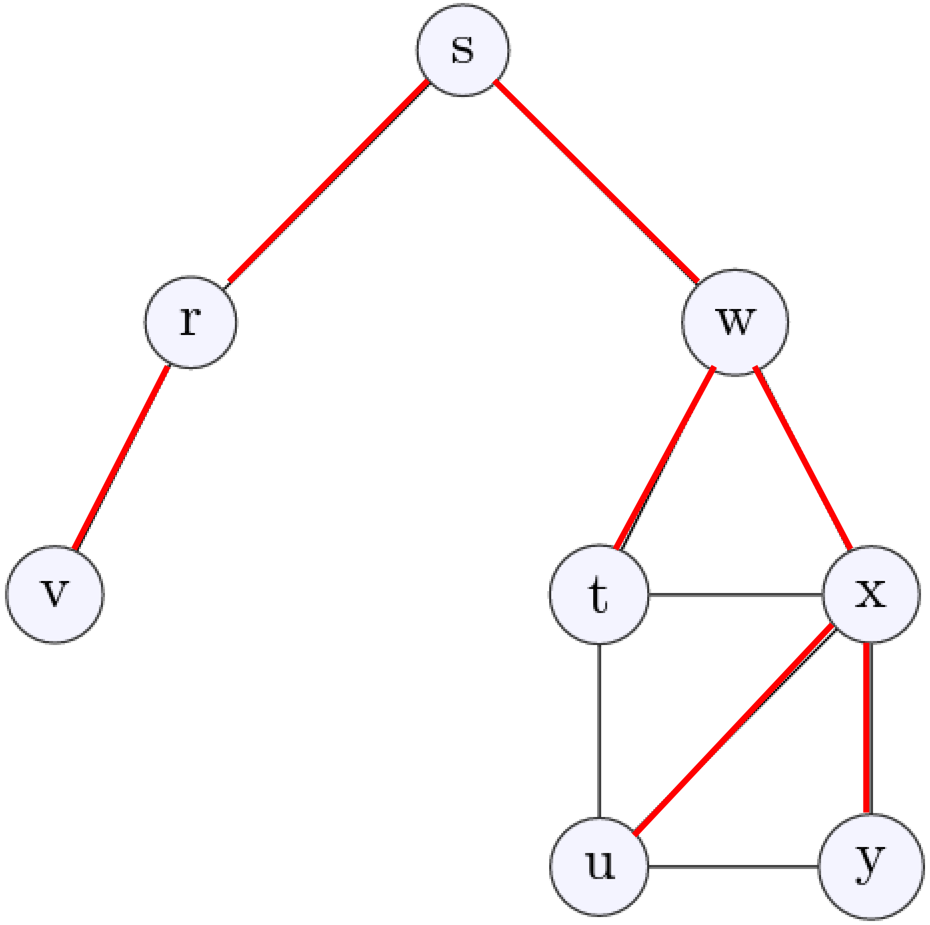


Figure 5.2: The Graph for Q3

Visiting neighbours in reverse alphabetical order

The sequences visited: s w r x t v y u

The queue Q: s w r x t v y u