Lecture 8 Oct 14, 13

Goals

- stack applications
    - convert from infix to postfix
- queues

Converting infix to Postfix expression

Recall the postfix notation from last lecture.

Example: a b c + *

It represents a*(b + c)

What is the postfix form of the expression a + b*(c+d)?

Answer: a b c d + * +

Observation 1: The order of operands in infix and postfix are exactly the same.

Observation 2: There are no parentheses in the postfix notation.

Example :

(a) Infix: 2 + 3 – 4 Postfix: 2 3 + 4 –

(b) Infix: 2 + 3 * 4 Postfix: 2 3 4 * +

The operators of the same priority appear in the same order, operator with higher priority appears before the one with lower priority.

Rule: hold the operators in a stack, and when a new operator comes, push it on the stack if it has higher priority. Else, pop the stack off and move the result to the output until the stack is empty or an operator with a lower priority is reached. Then, push the new operator on the stack.

Applying the correct rules on when to pop

Assign a priority: (* and / have a higher priority than + and – etc.)

Recall: Suppose st.top() is + and next token is *, then * is pushed on the stack.

However, (behaves differently. When it enters the stack, it has the highest priority since it is pushed on top no matter what is on stack. However, once it is in the stack, it allows every symbol to be pushed on top. Thus, its in-stack-priority is lowest.

We have two functions, ISP (in-stack-priority) and ICP (incoming-priority).

In-stack and in-coming priorities

	icp	isp
+, –	1	1
*, /	2	2
**	3	3
(	4	0

Dealing with parentheses

An opening parenthesis is pushed on the stack (always). It is not removed until a matching right parenthesis is encountered. At that point, the stack is popped until the matching (is reached.

Example: (a + b * c + d)* a

Stack: (	Output: a
Stack: (+	Output: a
Stack: (+	Output: a b
Stack: (+ *	Output: a b
Stack: (+ *	Output: a b c
Stack: (+	Output: a b c * +
Stack: (+	Output: a b c * + d
Stack	Output: a b c * + d +
Stack *	Output: a b c * + d +
Stack *	Output: a b c * + d + a
Stack	Output: a b c * + d + a *

Code for conversion (infix to postfix)

```
string postfix() {
        Stack st(100);
        string str ="";
        token tok = get_token();
        string cur = tok.get_content();
        while (tok.get_op_type() != -1) {
                if (tok.get_value() == 1)
                  str+= cur + " ";
                else if (cur == ")") {
                  while (st.Top()!= "(")
                     str += st.Pop() +" ";
                  string temp1 = st.Pop();
    }
```

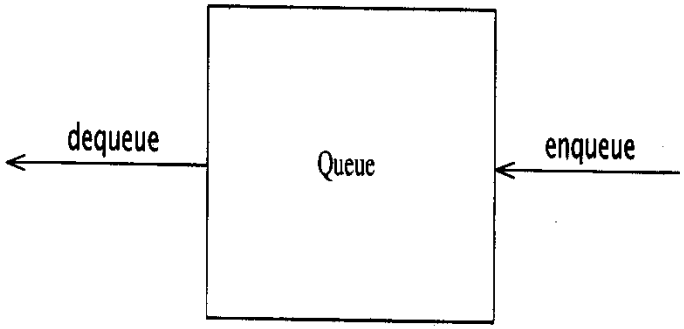
```
else if (!st.IsEmpty()) {
  string temp2 = st.Top();
  while (!st.IsEmpty() && icprio(cur) <= isprio(temp2)) {
        str+= temp2 + " ";
       string temp = st.Pop();
          if (!st.IsEmpty()) temp2 = st.Top();
}
            }
                 if (tok.get_value() != 1 && tok.get_content()!= ")")
st.Push(cur);
                 current++;
            tok = get_token();
            cur = tok.get_content();
     }
     while (!st.IsEmpty()) {
                 str += st.Pop() + " ";
                 cout << "string at this point is " << str << endl;
      }
     return str;
    }
```

Queue Overview

- Queue ADT
  - FIFO (first-in first-out data structure)
- Basic operations of queue
  - Insert, delete etc.
- Implementation of queue
  - Array

Queue ADT

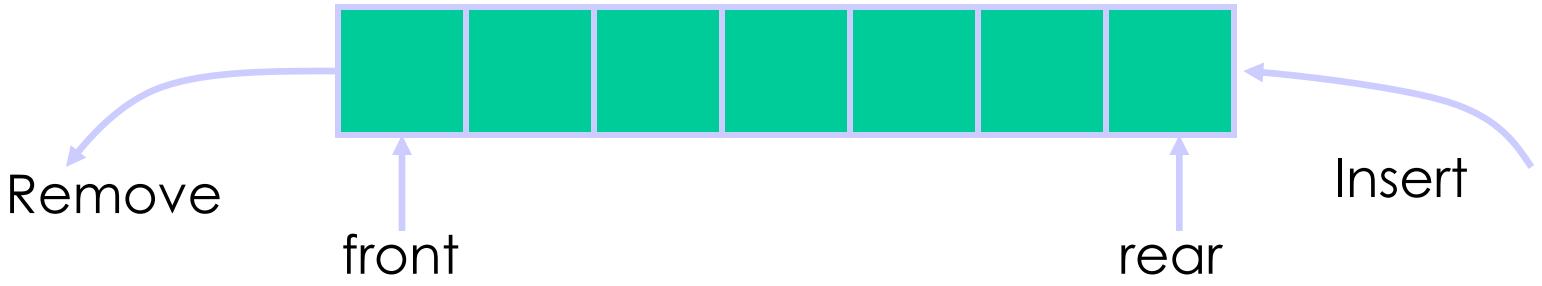
- Like a stack, a *queue* is also a list. However, with a queue, insertion is done at one end, while deletion is performed at the other end.



- Accessing the elements of queues follows a First In, First Out (FIFO) order.
  - Like customers standing in a check-out line in a store, the first customer in is the first customer served.

Insert and delete

- Primary queue operations: insert and delete
- Like check-out lines in a store, a queue has a front and a rear.
- *insert* - add an element at the rear of the queue
- *delete* – remove an element from the front of the queue



Implementation of Queue

- Queues can be implemented as arrays

- Just as in the case of a stack, queue operations (**`insert delete, isEmpty, isFull`**, etc.) can be implemented in constant time

Implementation using Circular Array

- When an element moves past the end of a circular array, it wraps around to the beginning, e.g.
  - OOOOO7963 → (insert(4)) 4OOOO7963

- How to detect an empty or full queue, using a circular array?
  - Use a counter for the number of elements in the queue.
  - size == ARRAY_SIZE means queue is full
  - Size == 0 means queue is empty.

Queue Class

- Attributes of Queue
  - front/rear: front/rear index
  - size: number of elements in the queue
  - Q: array which stores elements of the queue

- Operations of Queue
  - **`IsEmpty()`**: return true if queue is empty, return false otherwise
  - **`IsFull()`**: return true if queue is full, return false otherwise
  - **`Insert(k)`**: add an element to the rear of queue
  - **`Delete()`**: delete the element at the front of queue

```
class queue {
      private:
        point* Q[MSIZE];
        int front, rear, size;

     public:
      queue() {
      // initialize an empty queue
       front = 0; rear = 0; size = 0;
        for (int j=0; j < MSIZE; ++j)
          Q[j] = 0;
 }

  void insert(point* x) {
   if (!isFull()) {
          rear++; size++;
          if (rear == MSIZE) rear = 0;
          Q[rear] = x;
    }
 }
```

```
point* delete() {
  if (!isEmpty()) {
        front++; if (front == MSIZE) front = 0;
        size--;
        return Q[front];
   };
 }
bool isEmpty() {
  return (size == 0);
 }

bool isFull() {
  return (size == MSIZE);
  }
};
```

Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

Breadth-First Search

- Again will associate vertex “colors” to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

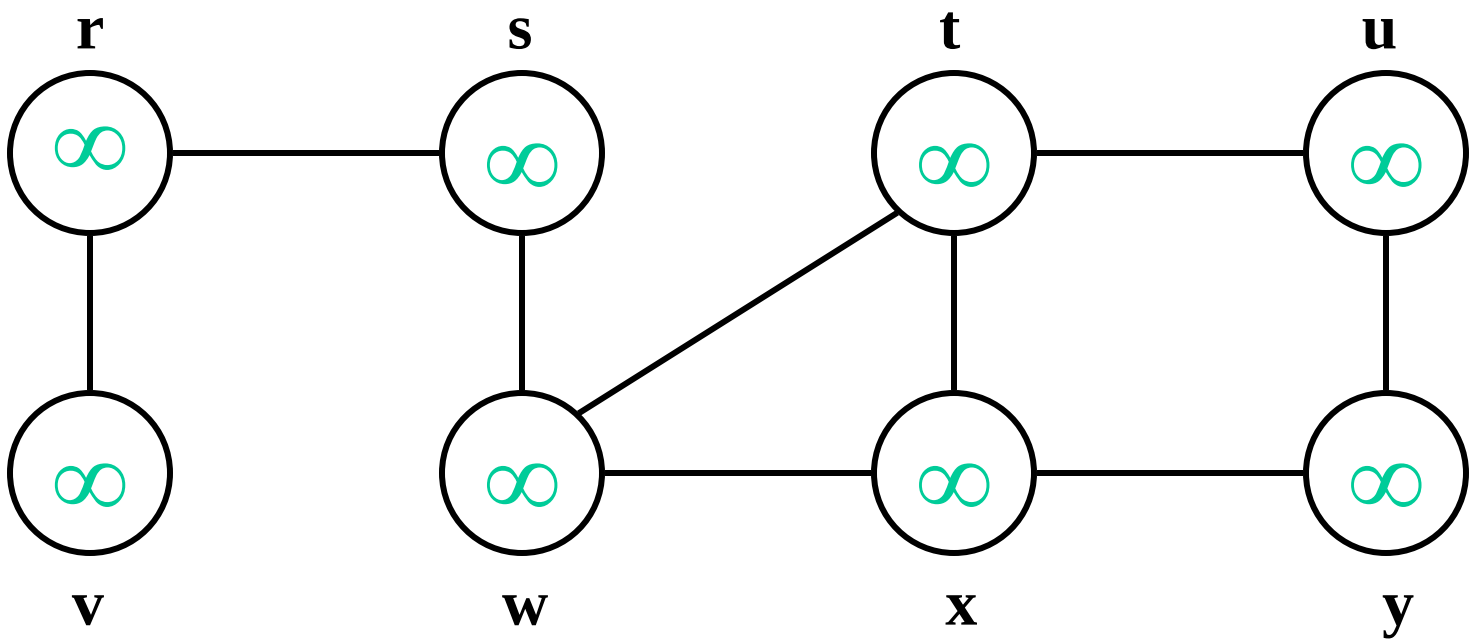
Review: Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};            // Q is a queue; initialize to s
    color all vertices WHITE; set the color of s BLACK;
    set d[s] = 0 and d[u] = MaxInt for all other u.
    while (Q not empty) {
        u =  Q->delete();
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = BLACK;
                d[v] = d[u] + 1;
                p[v] = u;
                Q->insert(v);
        }
   }
}
```

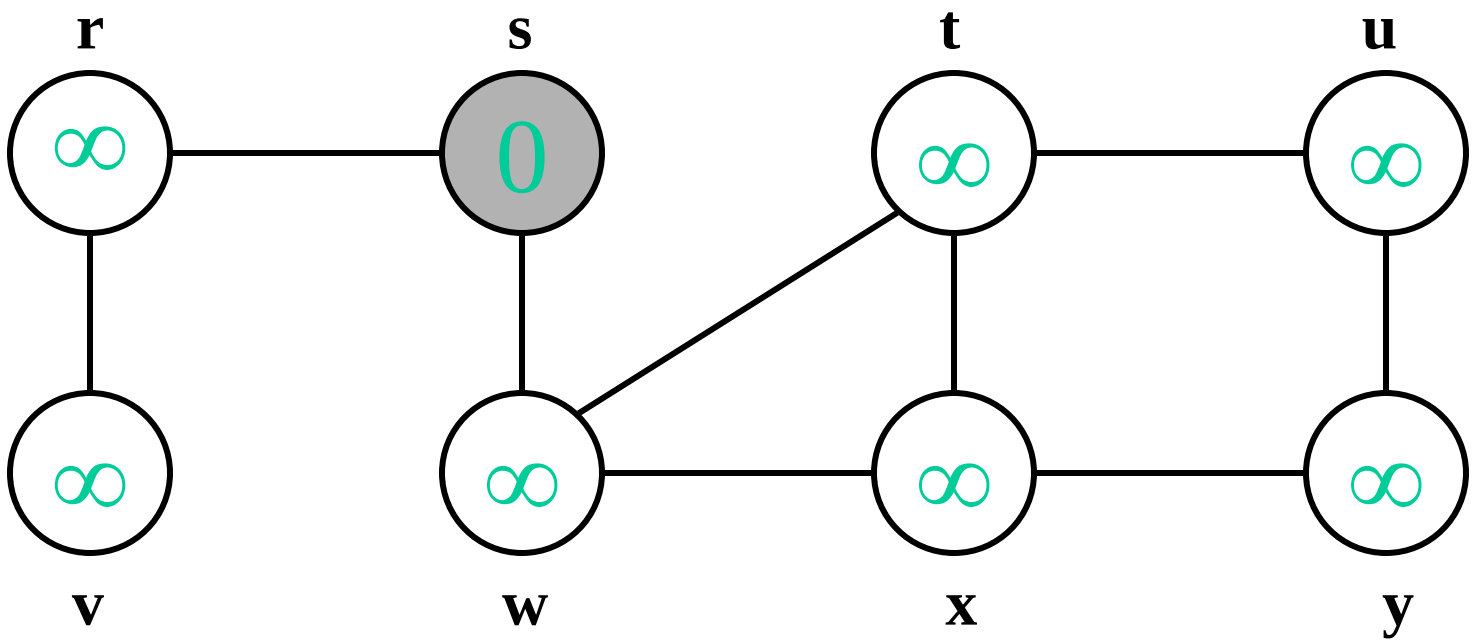
What does `v->d` represent?

What does `v->p` represent?

Breadth-First Search: Example

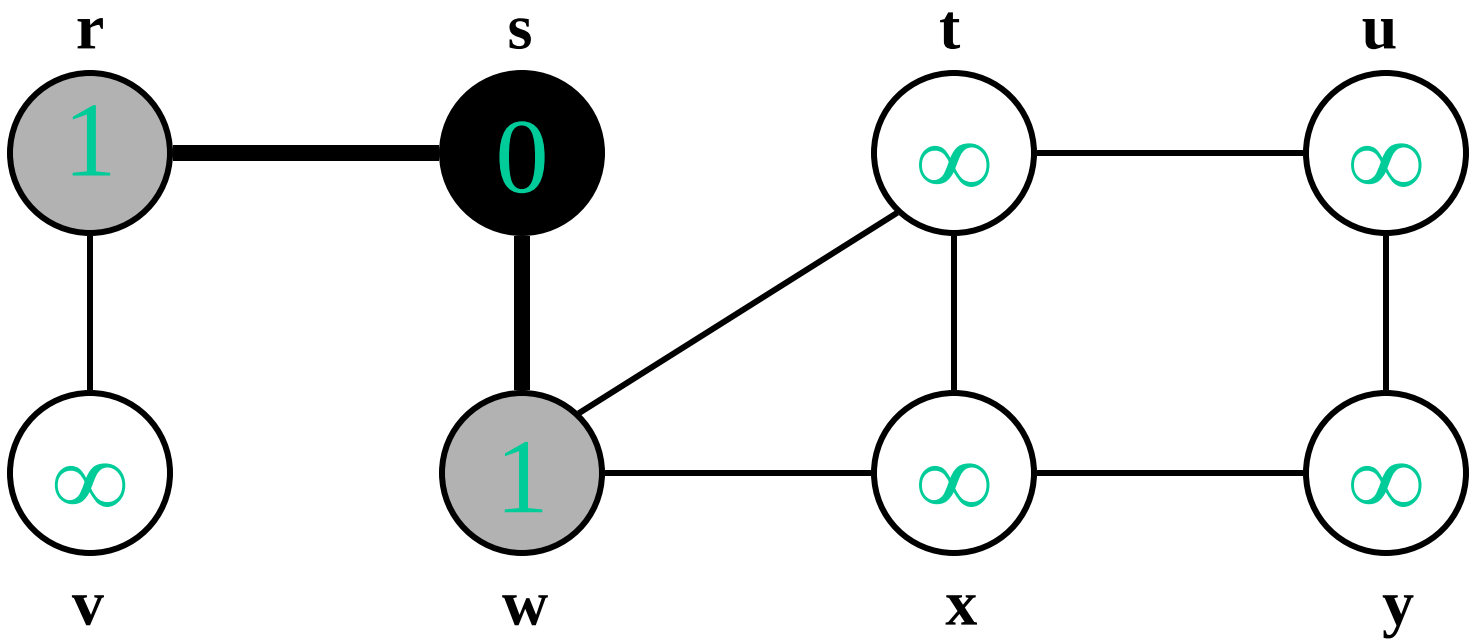


Breadth-First Search: Example



Q: s

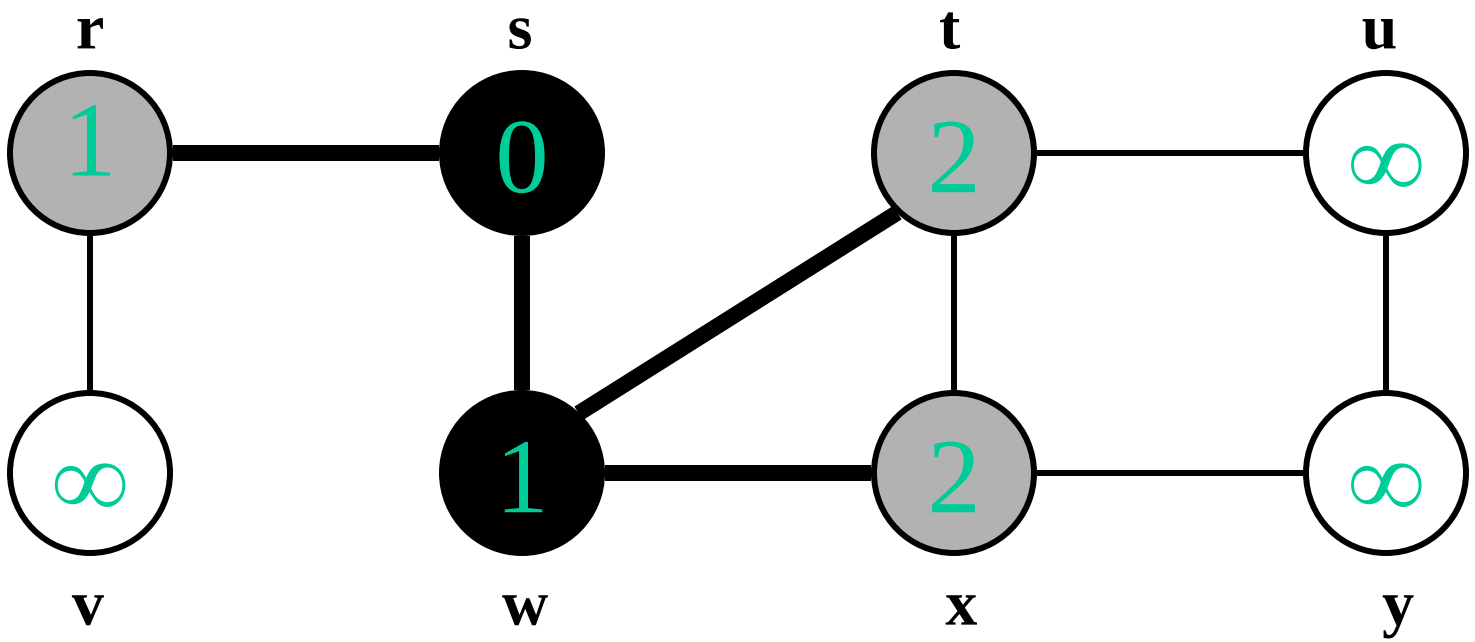
Breadth-First Search: Example



Q:

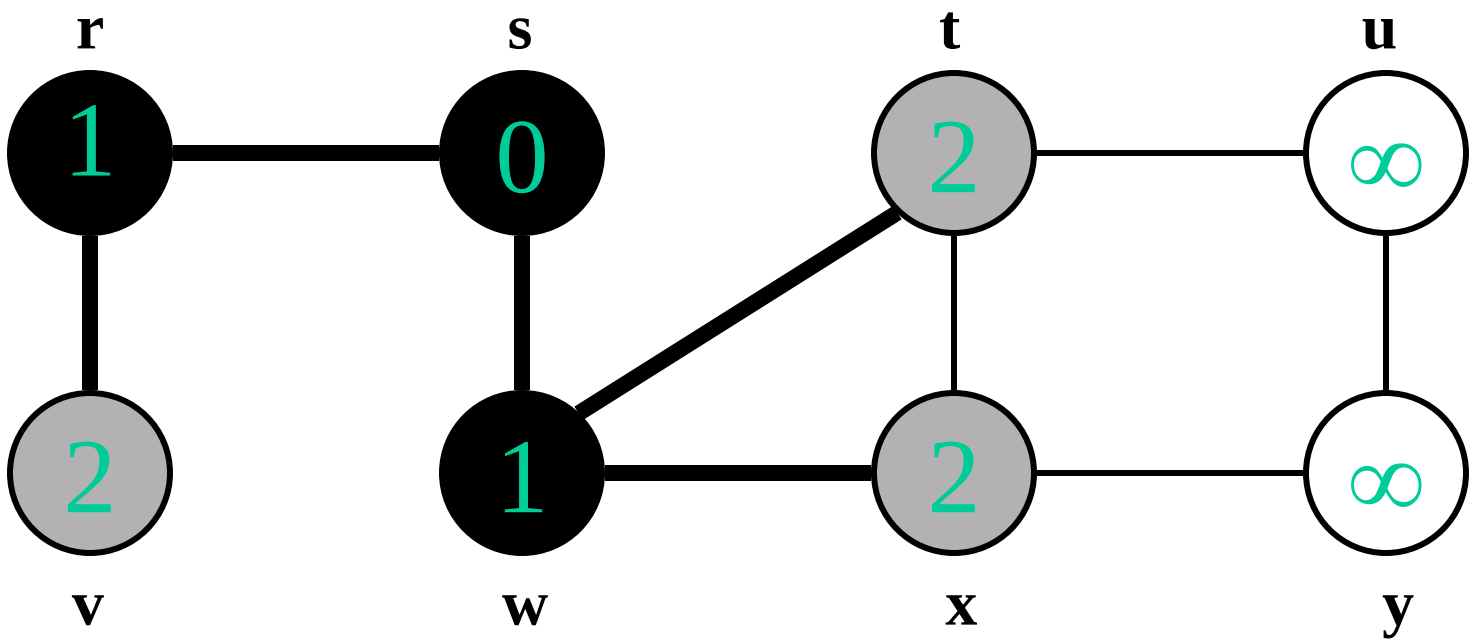
w	r

Breadth-First Search: Example



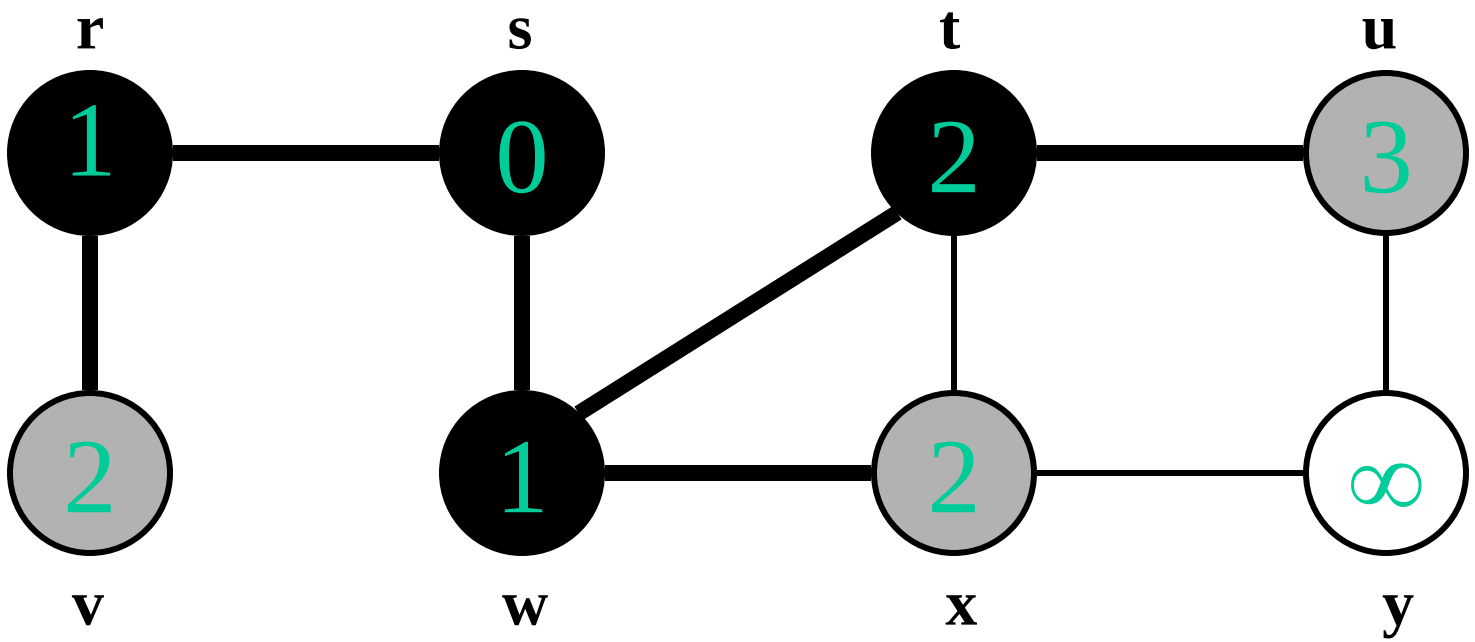
Q: | **r** | **t** | **x** |

Breadth-First Search: Example



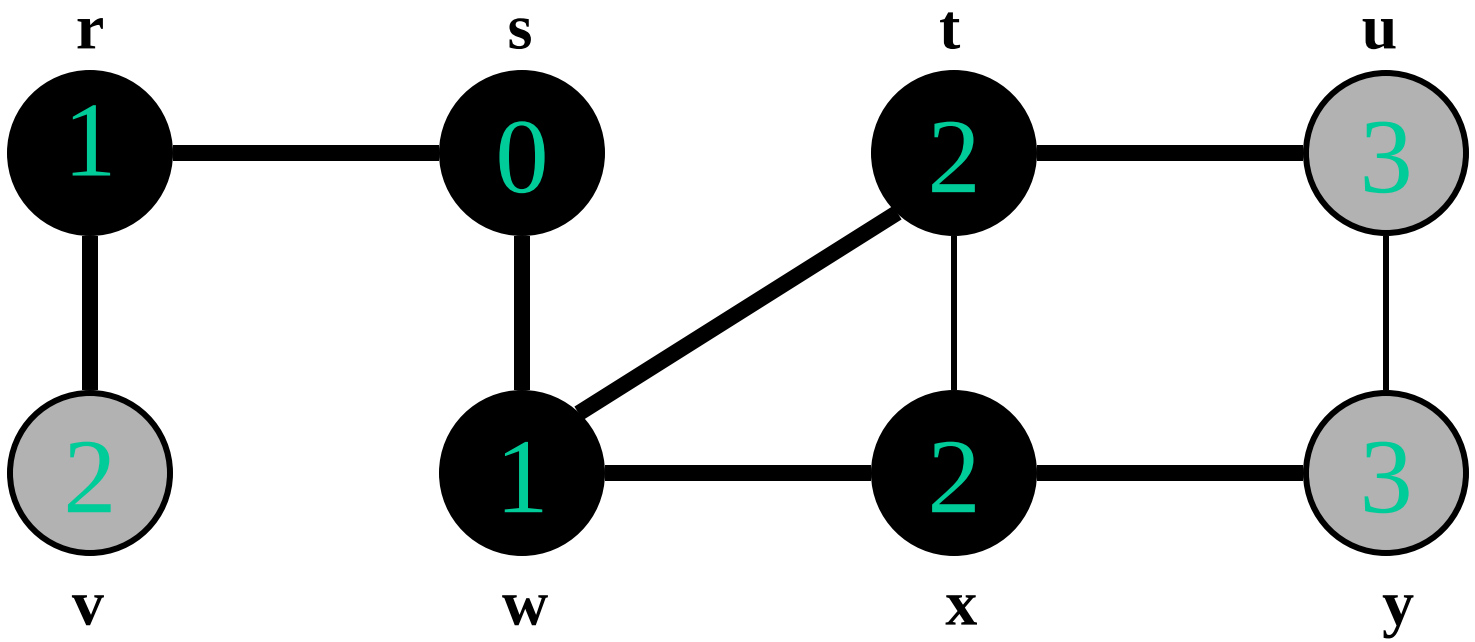
Q: | **t** | **x** | **v** |

Breadth-First Search: Example



Q: | **x** | **v** | **u** |

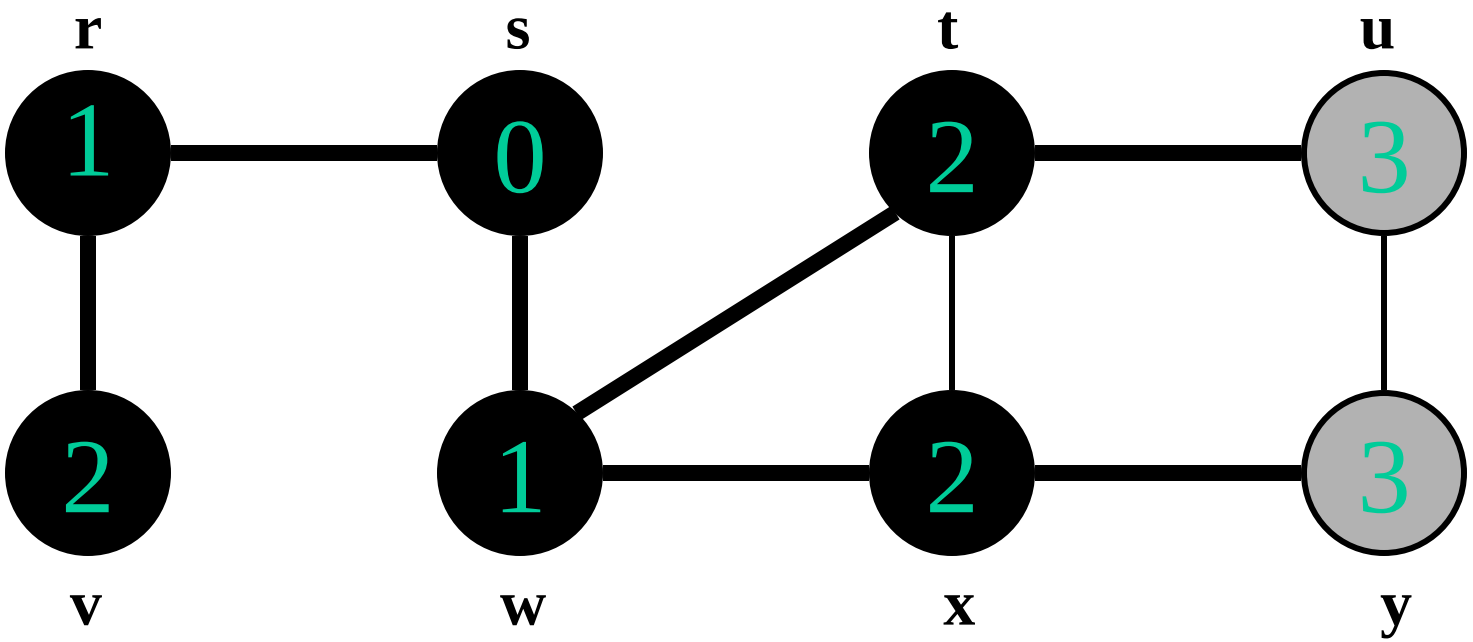
Breadth-First Search: Example



Q:

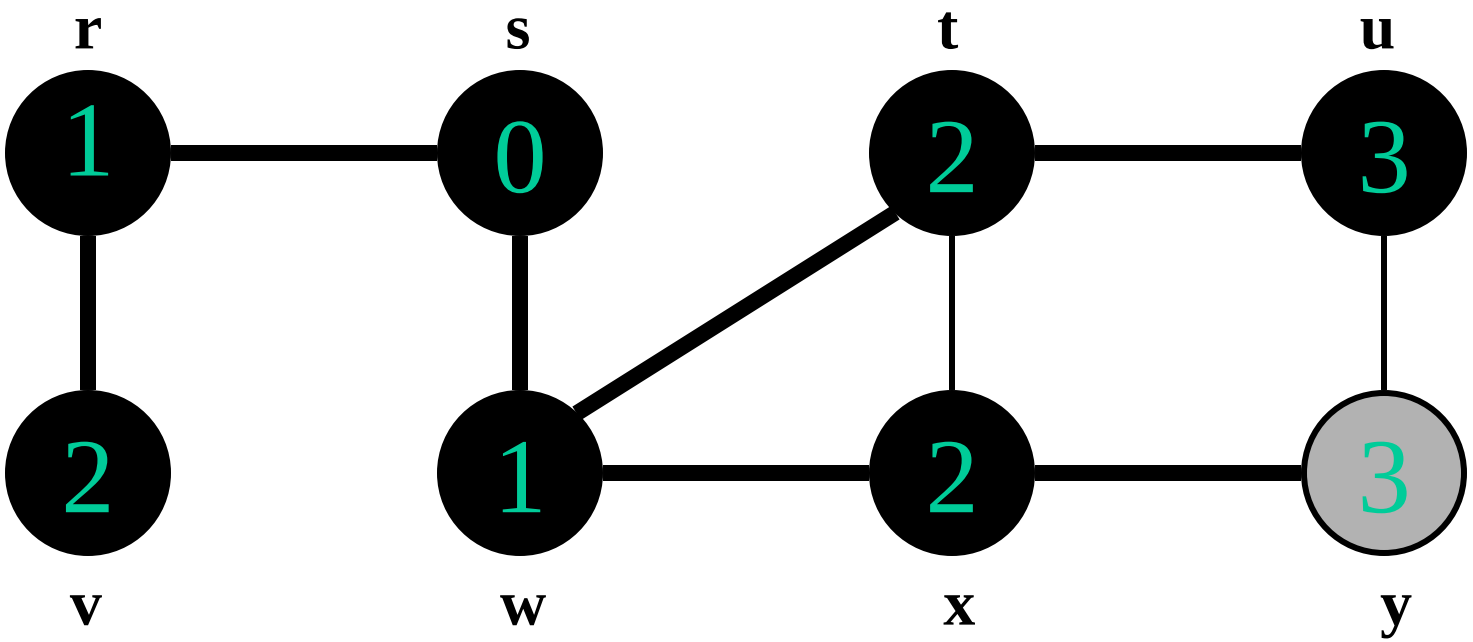
v	u	y

Breadth-First Search: Example



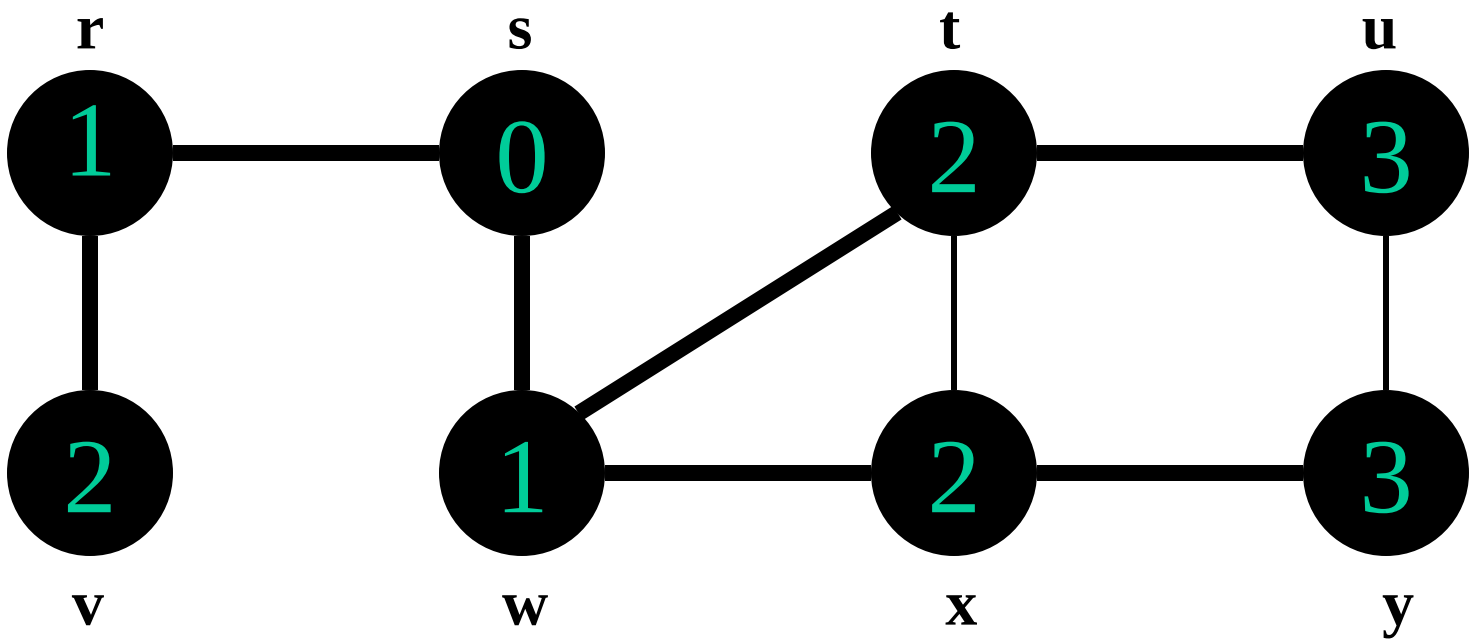
Q: | **u** | **y** |

Breadth-First Search: Example



Q: y

Breadth-First Search: Example

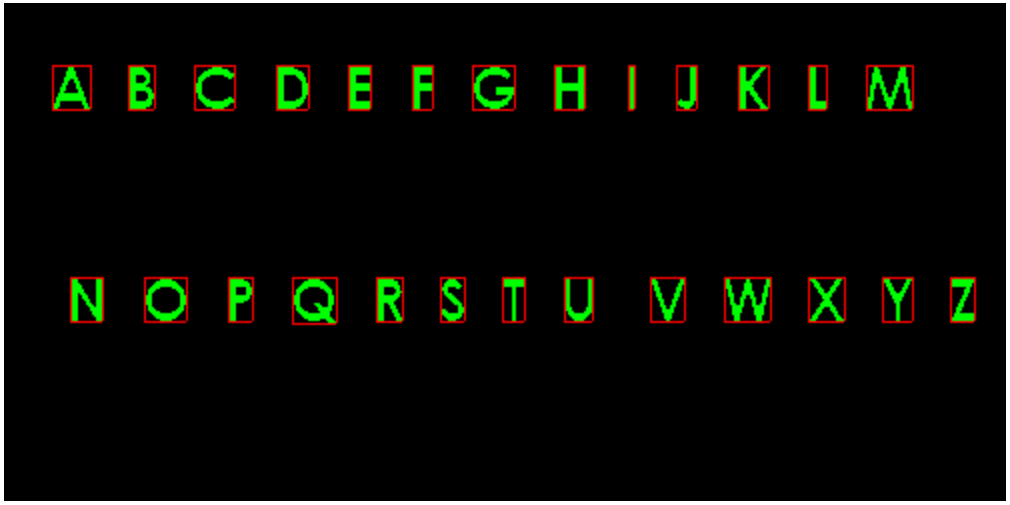


Q: Ø

Breadth-first search to count number of letters

BFS: application that can be implemented using a queue.

Our application involves finding the number of distinct letters that appear in an image and draw bounding boxes around them.



Taken from the output of the BFS algorithm