Algorithms and Data Structures

Module 1

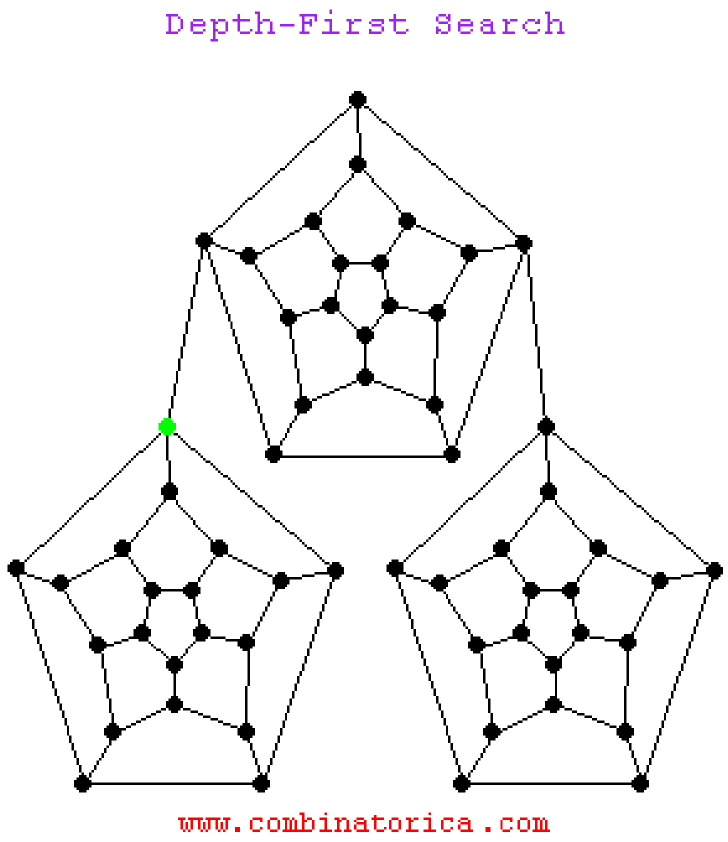
Lecture 5

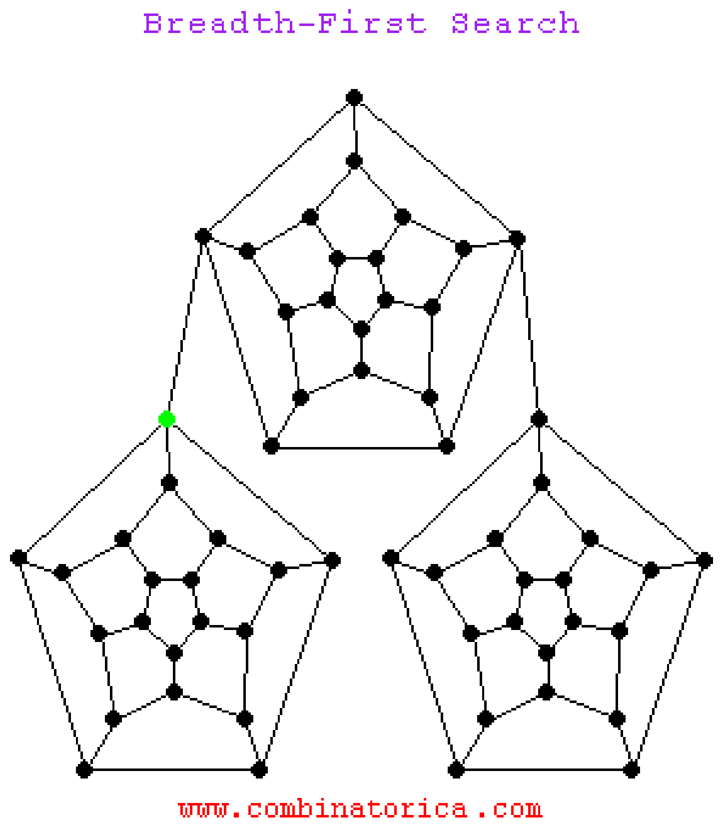
Graph traversals: depth-first search, breadth-first search and their applications. Part 2

Adigeev Mikhail Georgievich

mgadigeev@sfedu.ru

Graph traversals





https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

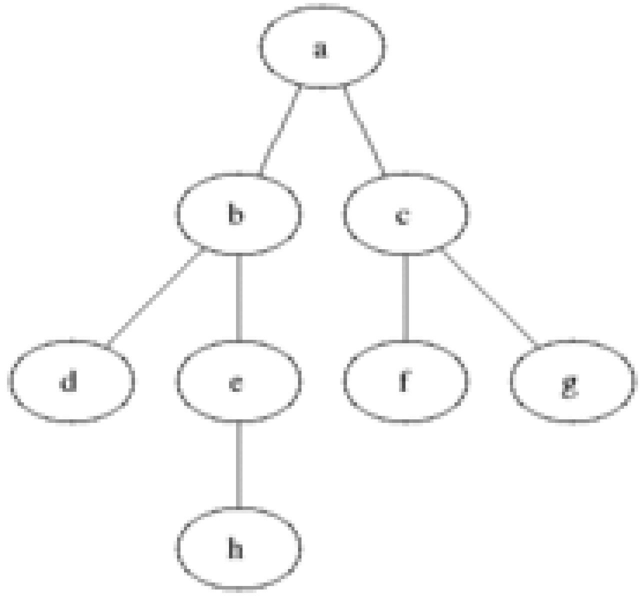
BFS: Breadth-First Search

Visiting a vertex v,

visit each of its unvisited neighbors,

then neighbors of the neighbors,

etc.

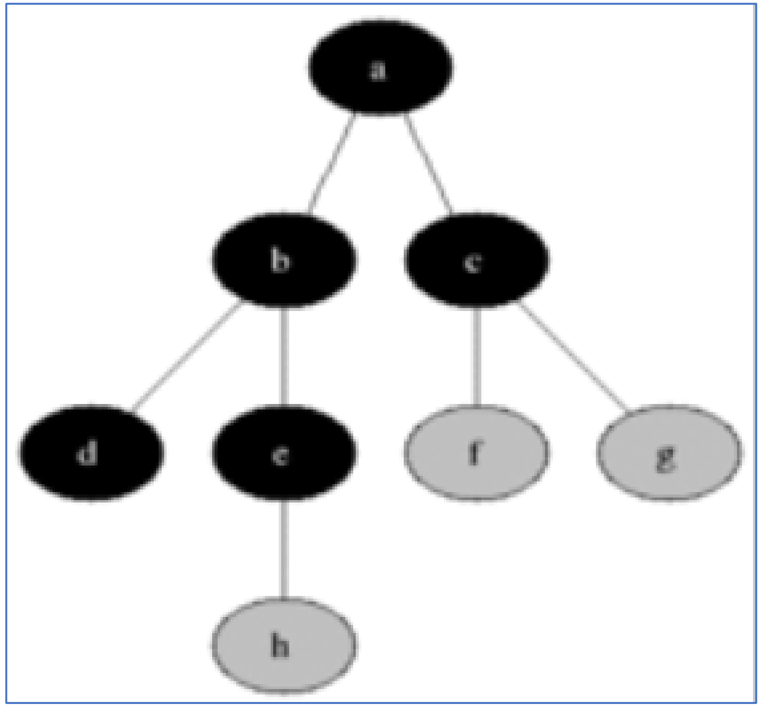


https://en.wikipedia.org/wiki/Breadth-first_search

BFS: Breadth-First Search

For keeping this order of visiting, we need to store neighbor vertices until we get them for processing.

We need a queue.



https://en.wikipedia.org/wiki/Breadth-first_search

Queue: abstract data structure

Queue = abstract data structure with two principal operations:

- Enqueue(*item*)
- Dequeue()

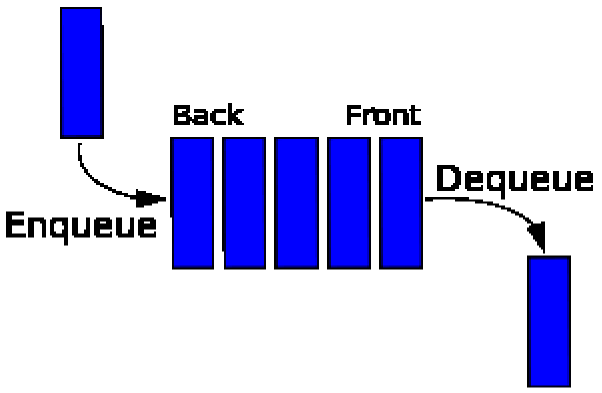
https://www.javascripttutorial.net/javascript-queue/

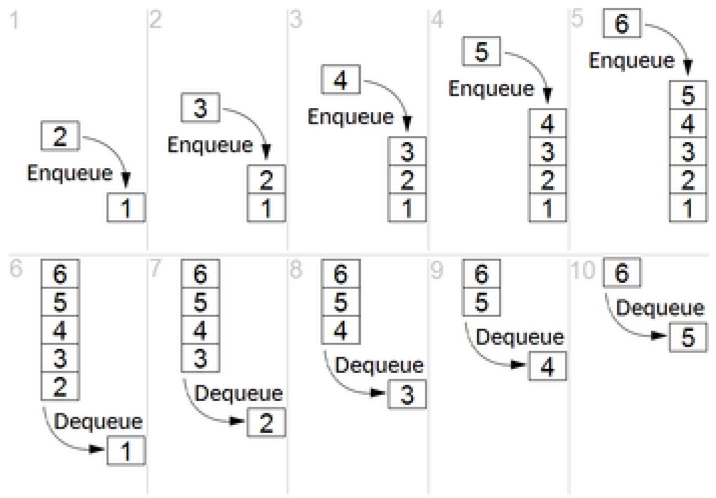
FIFO = First-In, First-Out

Queue: abstract data structure

Queue = abstract data structure with two principal operations:

- Enqueue(*item*)
- Dequeue()





FIFO = First-In, First-Out

https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)

Queue: abstract data structure

```
// Abstract queue class
template <typename E> class Queue {
private:
  void operator =(const Queue&) {}     // Protect assignment
  Queue(const Queue&) {}          // Protect copy constructor

public:
  Queue() {}          // Default
  virtual ~Queue() {} // Base destructor

  // Reinitialize the queue.  The user is responsible for
  // reclaiming the storage used by the queue elements.
  virtual void clear() = 0;

  // Place an element at the rear of the queue.
  // it: The element being enqueued.
  virtual void enqueue(const E&) = 0;

  // Remove and return element at the front of the queue.
  // Return: The element at the front of the queue.
  virtual E dequeue() = 0;

  // Return: A copy of the front element.
  virtual const E& frontValue() const = 0;

  // Return: The number of elements in the queue.
  virtual int length() const = 0;
};
```

https://people.cs.vt.edu/~shaffer/Book/

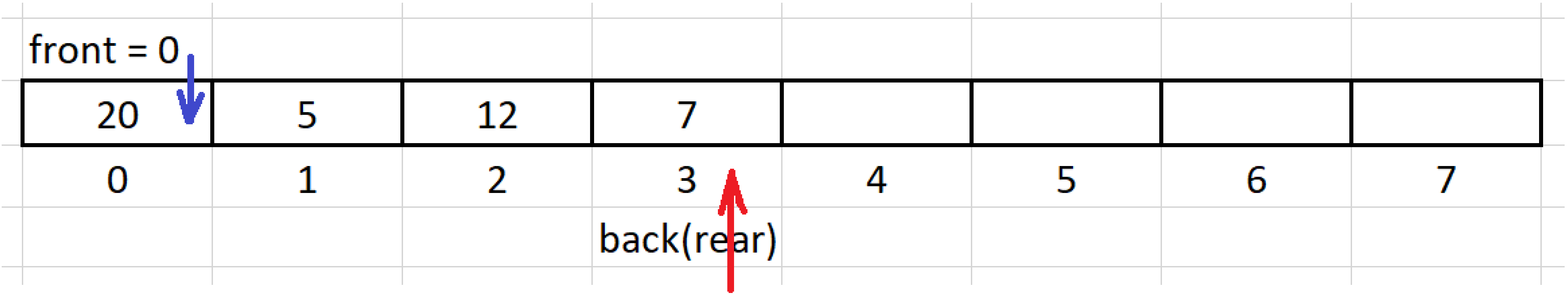
Queue: implementation

A queue data structure can be implemented in different ways:

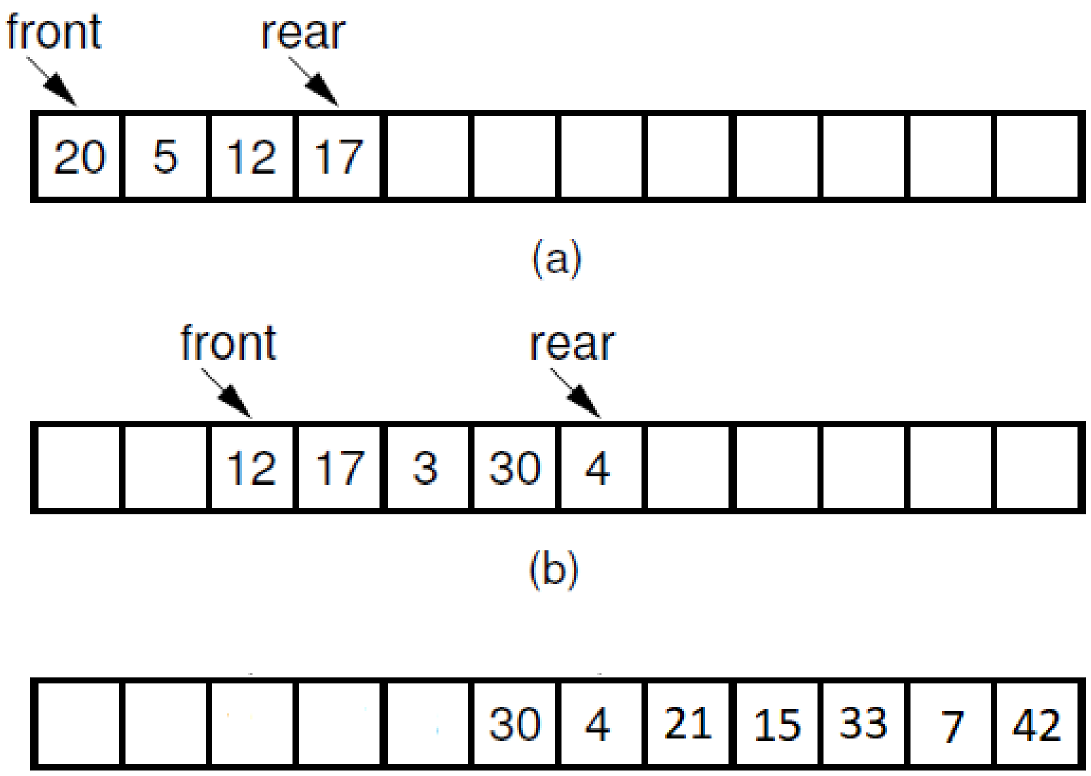
- Array-based
  - linear array
  - circular array
- Linked-list based

Queue: array-based implementation

Array-based implementation: keep indices of the front and the back(rear) items of the queue.

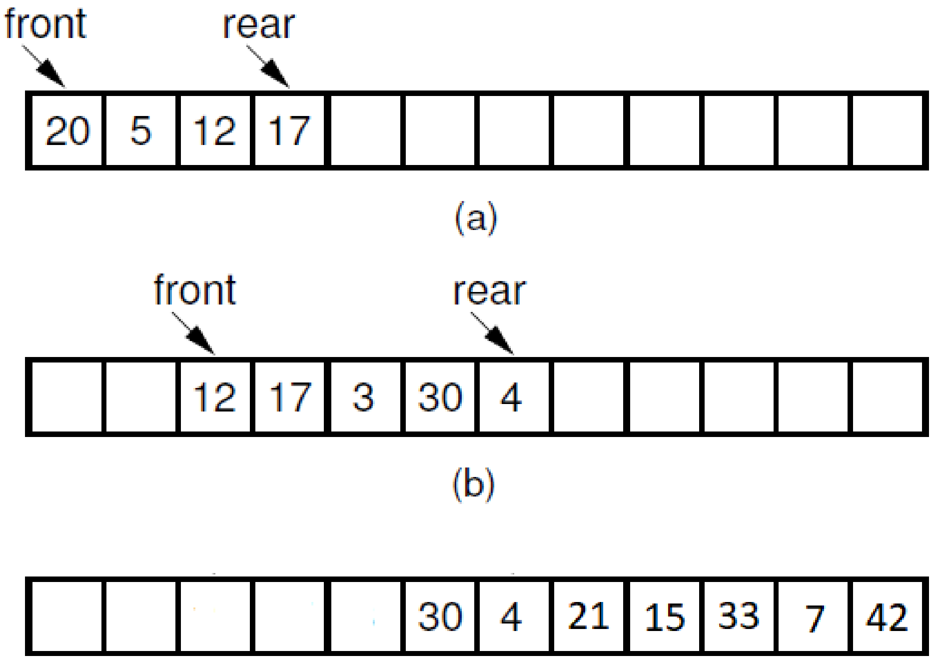


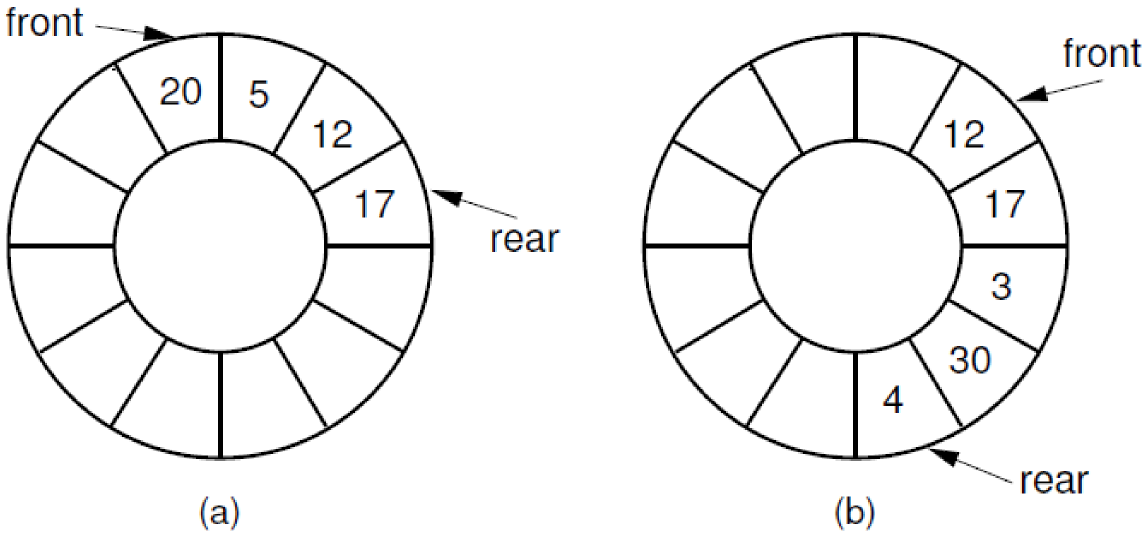
Queue: array-based implementation



https://people.cs.vt.edu/~shaffer/Book/

Queue: circular array-based implementation





https://people.cs.vt.edu/~shaffer/Book/

Queue: circular array-based implementation

Circular arrays implementation

We use an ordinary *linear* array (`E*` or `std::vector<E>`) and apply modular arithmetic when we increment / decrement indices.

Mathematical operation mod: 12 mod 10 = 2; 99 mod 10 = 9.

For an integer *x* and positive integer *m*, x mod m is an integer $y \in \{0, \ldots, m-1\}$ such that $x = y + km$ for some integer *k*.

In C++ we use % operation.

Queue: circular array-based implementation

Circular arrays implementation

```
void enqueue(const E& it) {      // Put "it" in queue
  Assert(((rear+2) % maxSize) != front, "Queue is full");
  rear = (rear+1) % maxSize;         // Circular increment
  listArray[rear] = it;
}

E dequeue() {              // Take element out
  Assert(length() != 0, "Queue is empty");
  E it = listArray[front];
  front = (front+1) % maxSize;    // Circular increment
  return it;
}
```

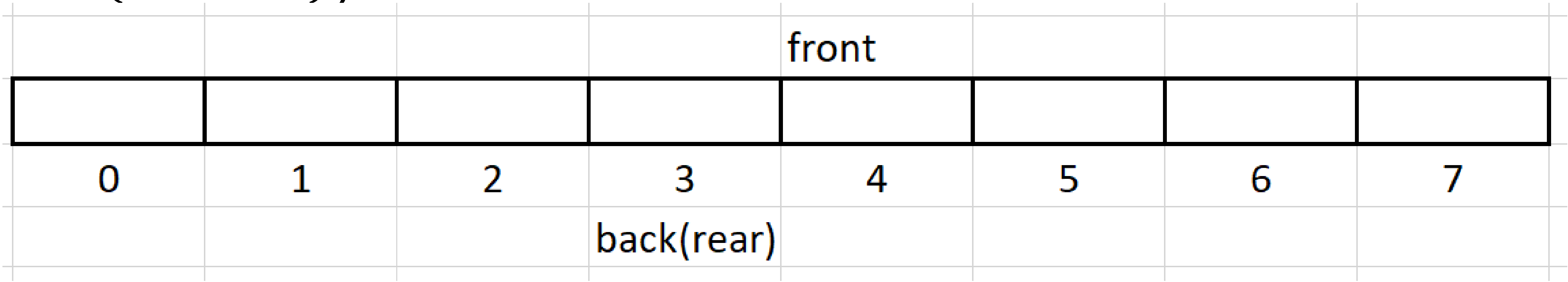
https://people.cs.vt.edu/~shaffer/Book/

Queue: circular array-based implementation

There is a potential problem with this implementation. Lets look at two cases:

a) Empty queue => the 'back' index is just before the 'front' index => $\text{back} = \text{front} - 1$.

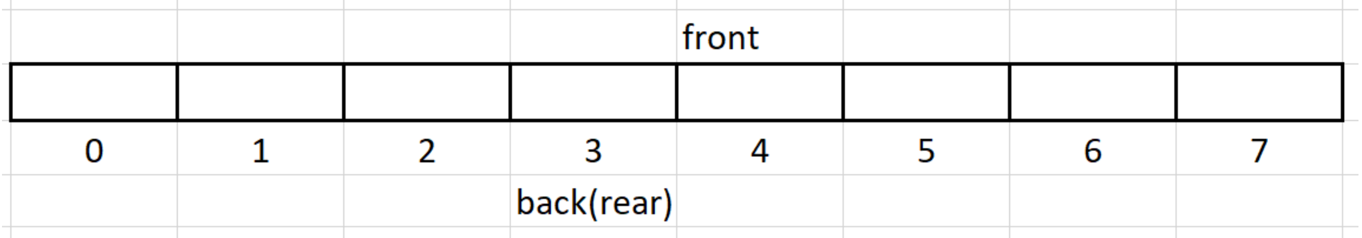
b) Full queue => $\text{back} = \text{front} + (size - 1)$ => $\text{back} = \left(\text{front} + (size - 1)\right) \,\%\, size$ => $\text{back} = \text{front} - 1$.



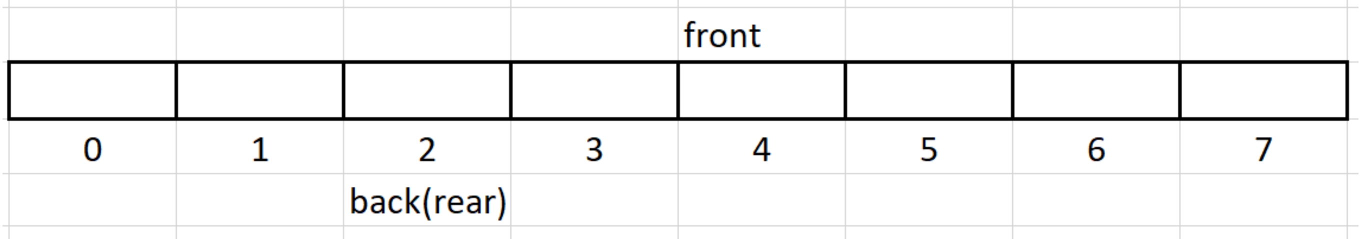
Queue: circular array-based implementation

Two possible solutions:

1) Keep an explicit count of the items in the queue:
   - Count = 0 => empty queue
   - Count = Size => full queue
2) Use array of size (n+1) for keeping maximum *n* items:
   - Empty queue:



   - Full queue:



Queue: circular array-based implementation

```
array<T> a;
int j;
int n;

bool add(T x) {
   if (n + 1 > a.length) resize();
   a[(j+n) % a.length] = x;
   n++;
   return true;
 }

T remove() {
  T x = a[j];
  j = (j + 1) % a.length;
  n--;
  if (a.length >= 3*n) resize();
  return x;
}

void resize() {
  array<T> b(max(1, 2*n));
  for (int k = 0; k < n; k++)
    b[k] = a[(j+k)%a.length];
  a = b;
    j = 0;
}
```

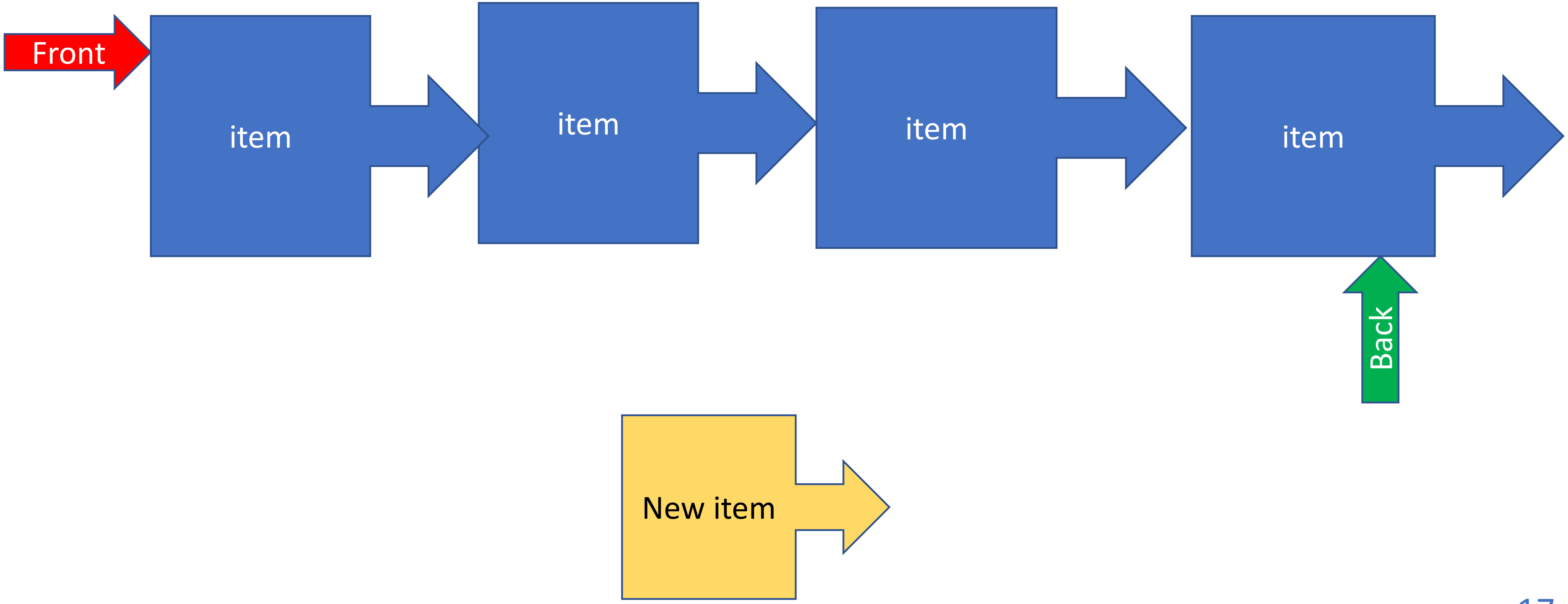
```
// Array-based queue implementation
template <typename E> class AQueue: public Queue<E> {
private:
  int maxSize;               // Maximum size of queue
  int front;                 // Index of front element
  int rear;                  // Index of rear element
  E *listArray;            // Array holding queue elements

public:
  AQueue(int size =defaultSize) {  // Constructor
    // Make list array one position larger for empty slot
    maxSize = size+1;
    rear = 0;  front = 1;
    listArray = new E[maxSize];
  }

  ˜AQueue() { delete [] listArray; } // Destructor

  void clear() { rear = 0; front = 1; } // Reinitialize

  void enqueue(const E& it) {      // Put "it" in queue
    Assert(((rear+2) % maxSize) != front, "Queue is full");
    rear = (rear+1) % maxSize;       // Circular increment
    listArray[rear] = it;
  }

  E dequeue() {            // Take element out
    Assert(length() != 0, "Queue is empty");
    E it = listArray[front];
    front = (front+1) % maxSize;     // Circular increment
    return it;
  }

  const E& frontValue() const {  // Get front value
    Assert(length() != 0, "Queue is empty");
    return listArray[front];
  }

  virtual int length() const          // Return length
   { return ((rear+maxSize) - front + 1) % maxSize; }
};
```

http://opendatastructures.org/

https://people.cs.vt.edu/~shaffer/Book/

Queue: dynamic list-based implementation

A dynamic list data structure with ‘front’ and ‘back’ pointers.



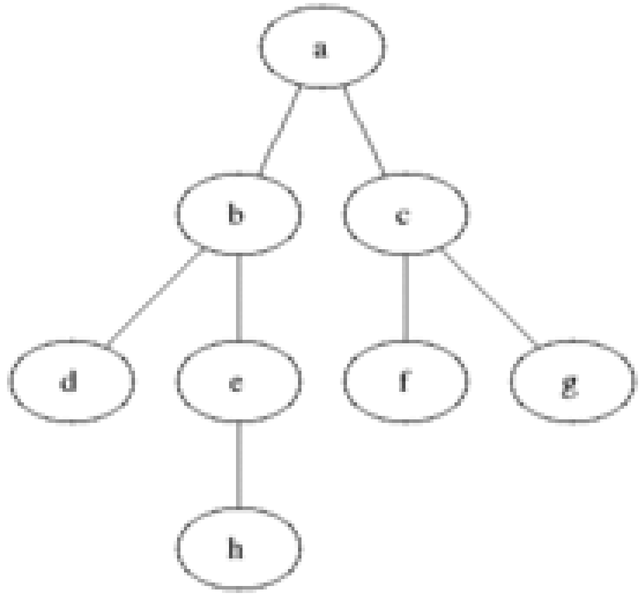
BFS: queue-based implementation

```
BFS(G)
Select s ∈ V
Enqueue(s)
While (Queue is not empty):
   v = Dequeue()
   if v is unvisited:
        Mark v as 'visited'
        For each u in Adj(v):
             Enqueue(u)
```

BFS: applications

1) Detecting connected components.
2) Calculating distances.

Principal idea: visiting a vertex v,
visit each of its unvisited neighbors,
then neighbors of the neighbors,
etc.



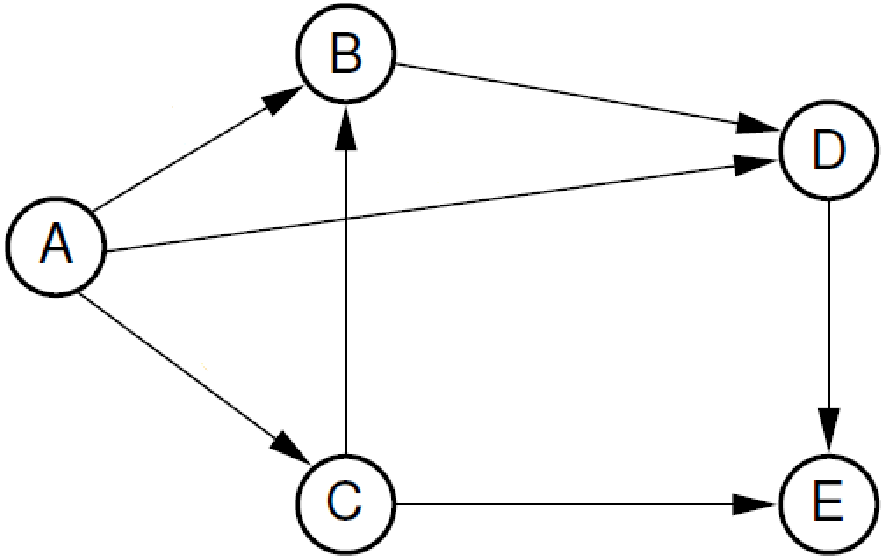
https://en.wikipedia.org/wiki/Breadth-first_search

BFS: applications

Graph G=(V,E).

A *distance* between vertices *u* and *v* is the minimum length of the path between *u* and *v*.

dist(A,E) = 2

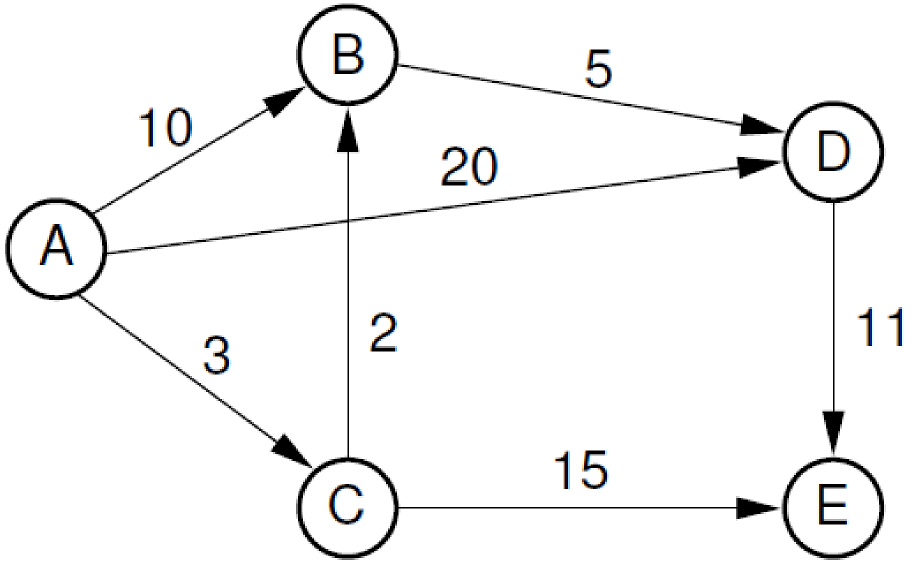


BFS: applications

Weighted graph G=(V,E), $w: E \rightarrow R$

A *distance* between vertices *u* and *v* is the minimum weight (=sum of edges' weights) of the path between *u* and *v*.

dist(A,E) = 18



BFS: applications

For unweighted graphs distances from $s \in V$ to all other vertices can be calculated using BFS.

For weighted graphs: Dijkstra algorithm works like a BFS and calculates distances (from $s \in V$ to all other vertices) on a graph.