DVA104
Data Structures, Algorithms, and Program Design.

ABU NASER MASUD

MASUD.ABUNASER@MDH.SE

Topics

- The `const` keyword
- ADTs:
  - ✓ Stack
  - ✓ Queue
  - ✓ Set
- Graph

Reading Materials

- ✓ Linked Lists: Introduction to Algorithms (Cormen) Pages relevant to the class lectures: 232-245, 589-611, 624-650)
- ✓ Book: Data structure using C, Reema Thareja (See PDF link in L3 slides)
  - Stack: PP 219-227, PP 230-232
  - For advanced readers: PP 233-250
  - Queue: PP 253-268
  - Graph: PP 383-400, 405-409, 413-414

*You may ignore materials that are not relevant to the class lectures

Pointers and Functions

Why do we send pointers to a function?

Pointers and Functions

Why do we send pointers to a function?

- Because we want to change the value that a pointer points in the function

```
void addCredits(Student *student, float credits);
```

- To avoid copying a struct

```
void foo(BigStruct * s);
```

- For arrays always sent as pointers

```
void printArray(int * A, size_t n);
```

- Because a data structure is represented by a pointer

```
void printList(Node * head);
```

The problem with pointers and functions

Why do we send pointers to a function?

- To avoid copying a struct
- For arrays always sent as pointers
- Because a data structure is represented by a pointer

In many of these cases, we do not want the function to change on what it points to.

Example:

```
/* Just print the array, do not change it */
void printArrayOnScreen(int* arr, int size);
/* We only want to return info from huge, do not change it*/
int getInfo(Hugestruct* huge);
/* List is a pointer, but the function may not change it */
 Data findElement(List list, Data element);
```

The problem with pointers and functions

- In these cases, we would like to leave the programmer a guarantee that the value being pointed out by the pointer will not be changed.
- The reason is that we want to avoid "surprises" or mistakes when calling a function.
- We can use the keyword **const** before a type name - then the compiler sees that the value pointed to by the pointer is not supposed to be changed.

The problem with pointers and functions

Better Examples:

```
/* The compiler now sees (by indicating errors) that the
elements in arrays can not be changed */
void printArrayOnScreen(const int* arr, int size);

/* The compiler ensures that no member of * huge can be
changed */
int getInfo(const Hugestruct* huge);


Data findElement(const List list, const Data element);
```

`Const` as Parameter

When to use a `const` parameter (rule of thumb)

- The parameter is a pointer
- What is pointed by the parameter should not be changed by the function
- When const should be used (example):

```
typedef Node* List;
bool findElement (const List list, const int item);
```

  - The list must not change (we will only find an item),
  - We may or may not use `const` for the second parameter

`Const` as Parameter

- When const should not be used (example)

```
void addFirst(List* list, int data);
```

In this case we have the post condition that `list` should point to a new item. That is, list has to be changed by the function.

const: watch out for the position!

```
const int *p = &n;
```

It is equivalent to:

```
int const *p = &n;
```

It is allowed to change p to point to any other integer variable, but we cannot change value of n by means of p.

```
int n = 10;
const int *p = &n;
*p = 20; //error: assignment of read-only location
p = NULL; // OK, no problem!
```

`const`: watch out for the position!

But, if we write:

```
int * const p = &n;
```

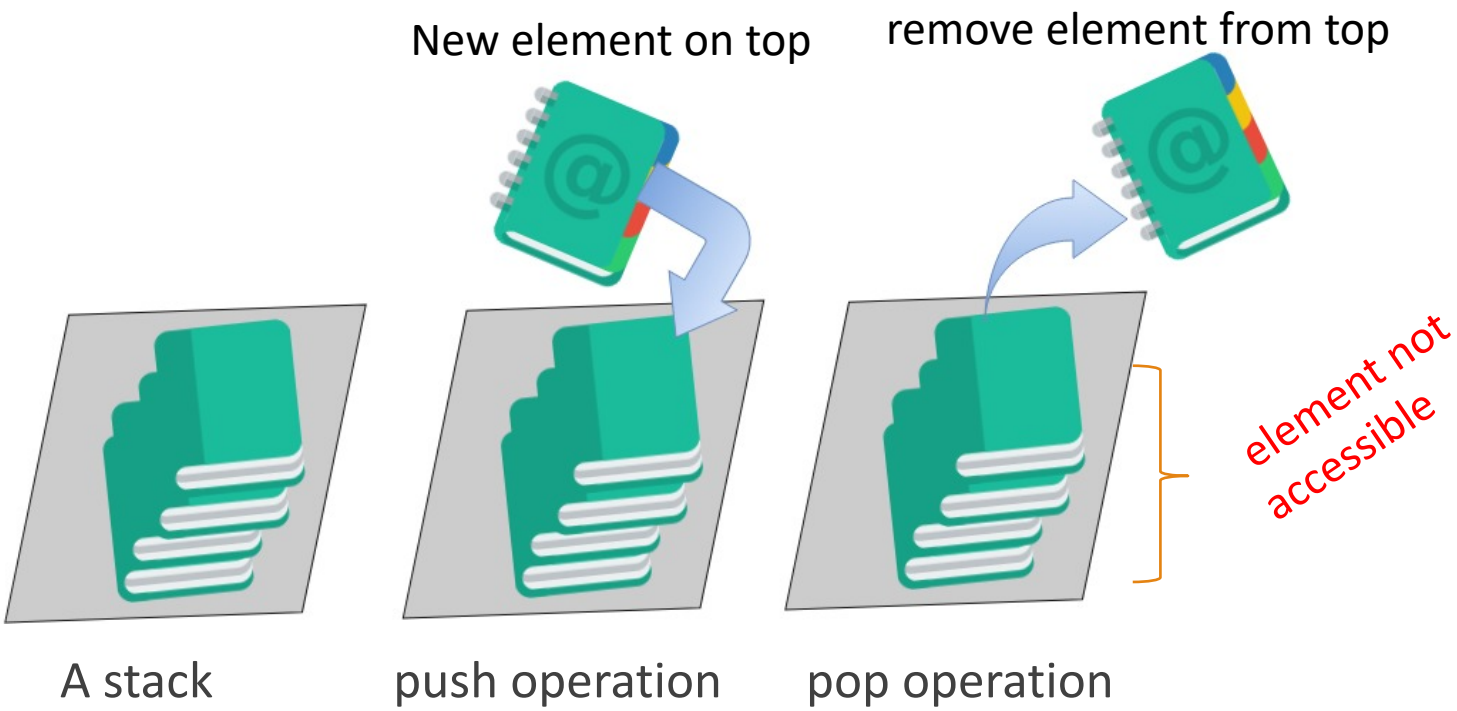
we get the opposite behaviour. That is we can change value pointed by p, but p cannot point another variable.

```
*p = 20; // OK, no problem!
 p = NULL; // error: assignment of read-only location
```

Stack

- A stack is a particular kind of abstract datatype (ADT). We can think of it as a logical structure consisting of
  - an **storag**e to store data of **homogeneous** types
  - Particular **operations** to manipulate the stored data
- Operations on stack:
  - The storage has a **top** and a **bottom**
  - New data can be **inserted** on top of the stack
  - Existing data can be **removed** from the top of the stack
  - Insertion operation is called **push** and remove operation is called **pop**

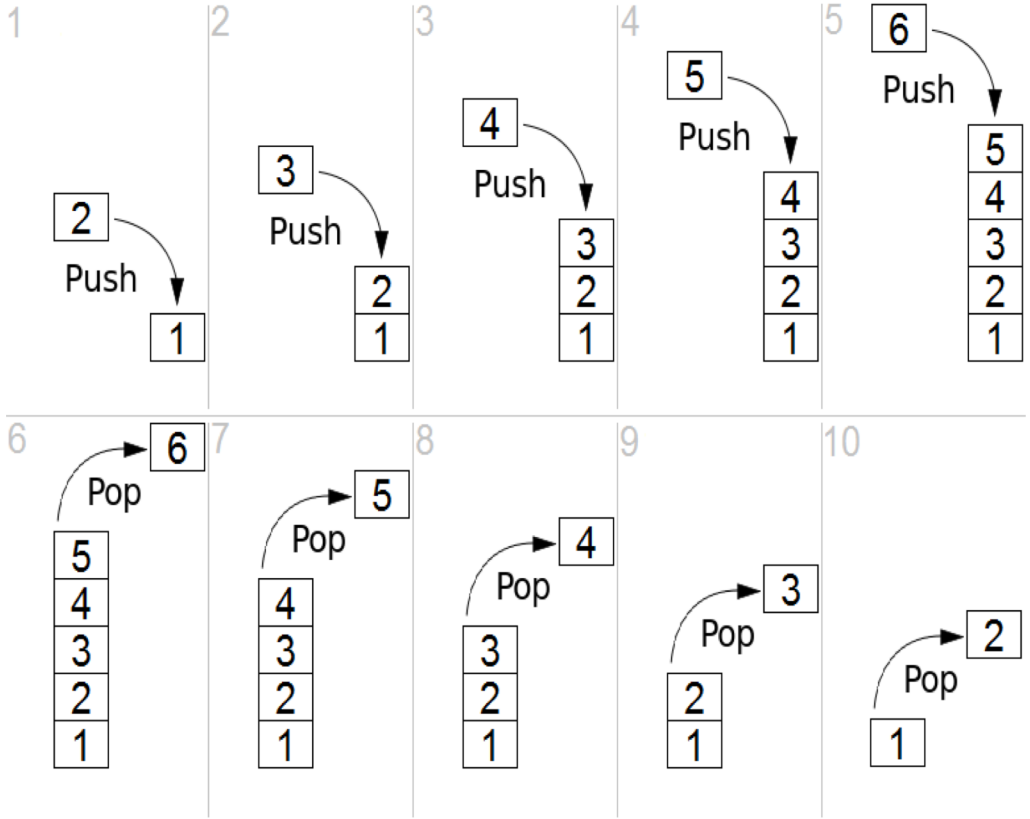
Stack Example



Stack Operations

- *Push* to add an item at the top.
- *Pop* to remove an item
- *Peek* looks at the top element

All elements in the stack should have the same but arbitrary type (A stack contains homogeneous data)



Stack Operations

```
/* Creates an empty stack */
Stack createStack (void);
```

Preconditions:

- None

Postconditions:

- Returns an empty stack

Stack operations

```
/* If the stack is empty, it returns 1,
 otherwise 0 */

int isEmpty(const Stack stack);
```

Preconditions:

- None

Postconditions:

- None

Stack operations

```
void push(Stack* stack, const Data data);
```

Preconditions:

- There is space for new element in the stack
  - Conceptually, there is no precondition. In practice, we should always check that there is enough space for storing the new item.

Postconditions:

- The value data is on the top of the stack.

Stack operations

```
void pop(Stack* stack);
```

Preconditions:

- The stack is not empty

Postconditions:

- The top item has been removed.

Stack operations

```
/* Return the value of the top item (but do
   not remove it from the stack)
*/

Data peek(const Stack stack);
```

Preconditions:

- The stack is not empty

Postconditions:

- None

Stack operations

```
/*Returns 1 if stack is full, 0 otherwise */
int isFull(const Stack stack);
```

Preconditions:

- None

Postconditions:

- None

Stack Interface

```
Stack createStack(void);
void push(Stack* stack, const Data data);
void pop(Stack* stack);
Data peek(const Stack stack);
int isEmpty(const Stack stack);
int isFull(const Stack stack);
```

Note:
To define an interface, we must first decide on the behavior of the ADT and its implementation strategy.

Stack: Usage example

```
#include "stack.h"
int main(void)
{
    /*We get a new stack*/
   Stack stack = createStack();
    /*Add element 5 on top of the stack*/
    push(&stack, 5);
    /*assertion: 5 must be on top of the stack*/
    assert(peek(stack) == 5);
    /*Remove the top element(5) from the stack*/
    pop(&stack);
    /*Assertion: stack must be empty*/
    assert(isEmpty(stack));
    return 0;
 }
```

Stack Exercise

```
#include "stack.h"
int main(void)
{
    Stack stack = createStack();
    push(&stack, 5);
    push(&stack, 8);
    push(&stack, 1);
    pop(&stack);
    pop(&stack);
    push(&stack, 3);
    pop(&stack);
    push(&stack, 1);
    pop(&stack);
    return 0;
}
```

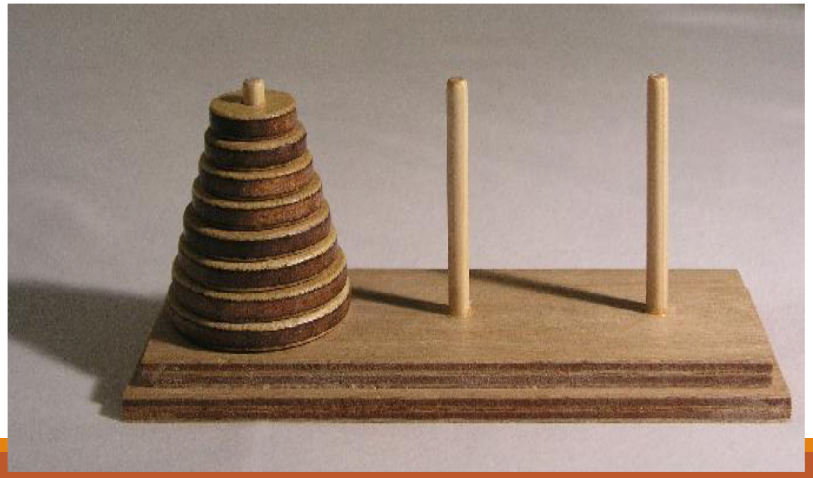
What is stored in the stack before the main returns?

Stack Application

- Minne allokeras som bekant “på stacken”
- Backtracking (undo)

En stack kan också kallas för en LIFO-kö (Last In First Out)

Towers of Hanoi
Varje “pinne” är en stack.



Solving Problems Using Stack:

Balanced parentheses

Problem:

Given a sequence of symbols containing parentheses, brackets, and braces, check that the sequence is balanced, i.e., every right brace, bracket, and parenthesis must correspond to its left counterpart.

Examples:

- a +b*[c-(d/e)*f] is balanced,
- a+[b-(c+d]*2) is not balanced

Solving Problems Using Stack:
Balanced parentheses

Algorithm
Input: sequence of characters SEQ
Output: return 'YES' if balanced, and 'NO' otherwise

- Create an empty stack.
- **While**(there are characters in the sequence) **Do**
  - CH = read_next_character(SEQ)
  - **If** *CH* is a left bracket (like (,{,[), **then** push it onto the stack
  - **If** *CH* is is a right bracket (like),},]) and the stack is empty, **then Return** NO
  - **Otherwise**, perform a pop operation.
    If the popped character does not correspond to CH, **then Return** NO

/* end while */

- **if** the stack is not empty, **then Return** NO
- **Return** YES

ADTs and Preconditions

- If you are calling an ADT function, **you** must ensure that the precondition is true! Otherwise, no guarantee about what the function does.

Example of incorrect use:

```
Stack stack = createStack();/*Create an empty stack*/
pop(&stack); /*Error! Precondition: stack cannot be
                empty*/
```

Calling `pop` on an empty stack is not a valid operation. It's up to the ADT function users to make sure it does not happen.

ADTs and Preconditions

Can we allow to remove an item if we call pop() on an empty stack?

- Sure, but we should avoid that the program crashes!
- However it doesn't fullfill the postcondition, that is after pop() the numbers of items in the stack should be decreased.
- We should consider cases like this one as a "division by 0" or "accessing the value of a NULL pointer".

Another example of incorrect usage:

```
Stack stack = createStack(); /*Create an empty stack*/
int x = peek(stack); /*Error! Stack cannot be empty */
```

What can peek() return? We cannot even return -1 (as error code) since -1 is a value that may be stored in the stack!

ADTs and Preconditions

To ensure that you do not violate preconditions:

- Avoid calling functions incorrectly and test the program properly using assert.
- In some cases, preconditions may need to be verified before:

```
/* The stack may or may not be empty,so we check
   first */
if(isEmpty(stack))
  printf("Stack is empty!");
else
      pop(&stack); /*Stack is not empty*/
```

Implementation of ADTs

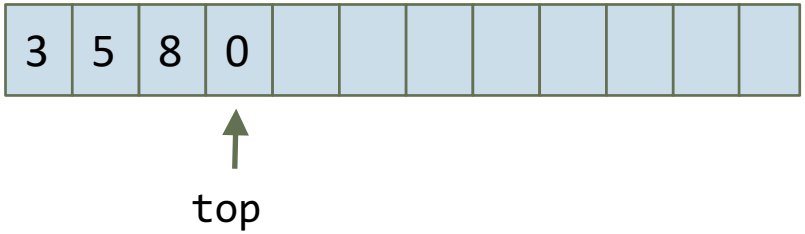
- We have now looked at the functions specifying the behaviour of Stack. However, we do not know yet
  - how the stack is implemented,
  - what data type / structure it is
- A stack can be implemented in different ways
  - Array
  - Linked list

Array-based Implementation of Stack

Let us see how to implement a stack by means of an array.

We need the followings:

- An array to store the stack elements
- A variable which keeps track of what's the top of the stack.



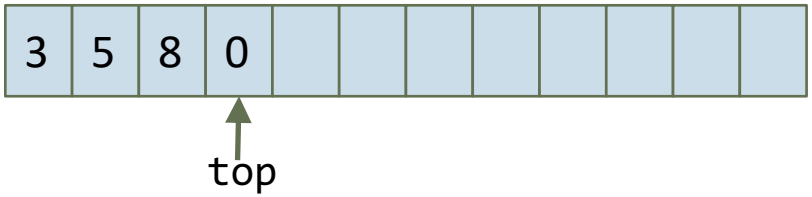
Array-based Implementation of Stack

- We can thus represent the stack as a structure containing an array to hold data, and a variable for the top index of the stack:

```
struct arraystack
{
    Data arr[MAXLENGTH];
    int top;
};
typedef struct arraystack Stack;
```

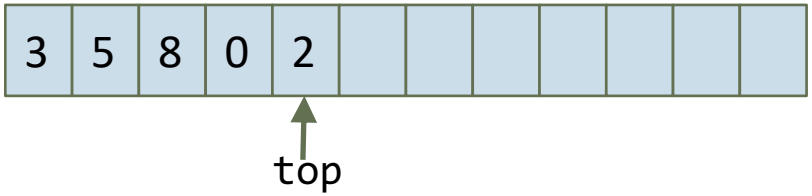
- The stack consists of the elements between index 0 and index top.

Array-based Implementation of Stack



After the operation

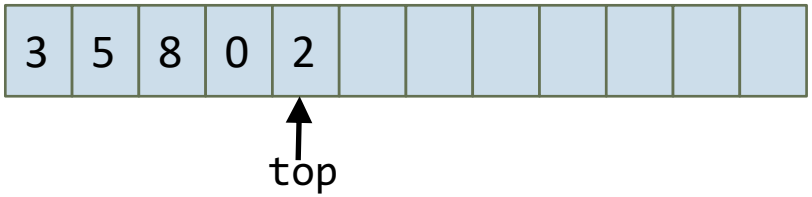
```
push(&stack, 2);
```

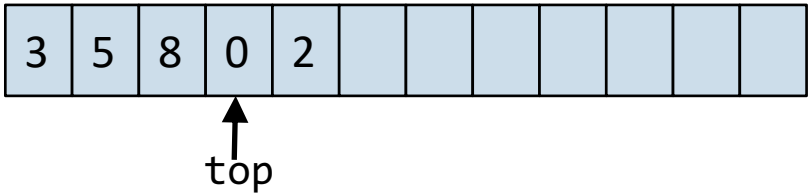


Algorithm:

1. Check that stack is not full
2. Increment `top` by 1
3. Add the element

Array-based Implementation of Stack





After the operation

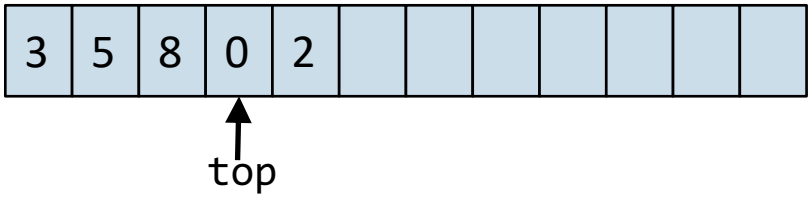
```
pop(&stack);
```

Algorithm:

1. Check that stack is not empty.
2. decrement top by 1

It does not matter that 2 is in the array.

Array-based Implementation of Stack



Operatiom

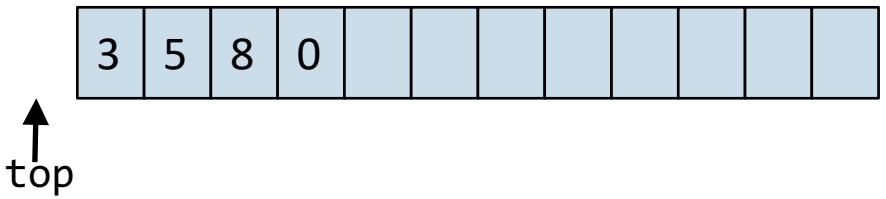
```
peek(stack);
```

Returns `0`

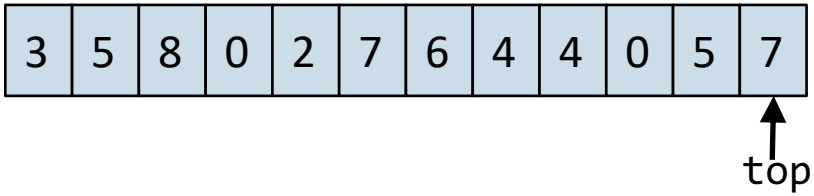
Algorithm:

1. Check that stack is not empty
2. Return the top element

Array-based Implementation of Stack



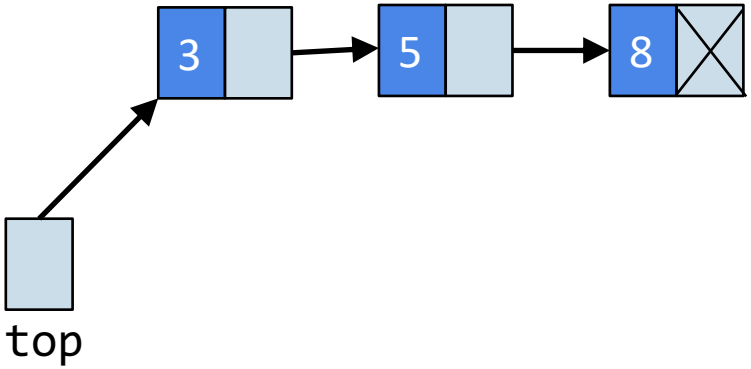
If stack is *empty*, then `top` is `-1`



If stack is *full* then `top` is `MAXLENGTH-1`

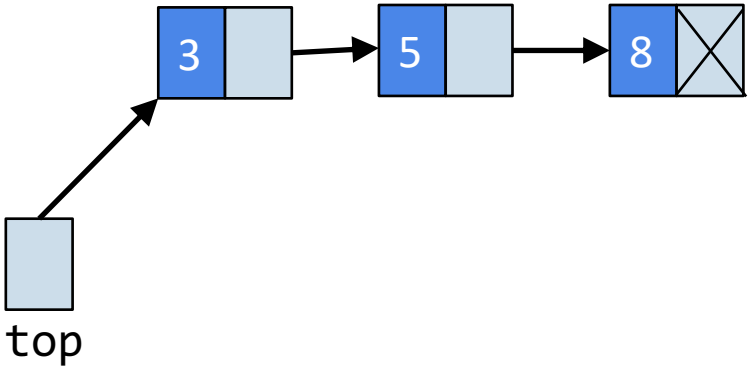
`MAXLENGTH` is the maximum length of the array.

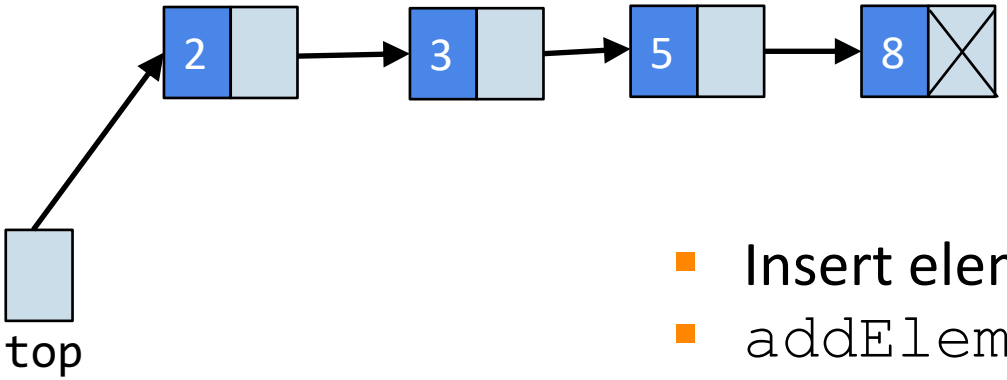
Linked list-based Implementation of Stack



- The list head will be the top of the stack
- We can define stack as follows:
  - **`typedef`** `List Stack;`
  - Now, stack can be implemented as a linked list

Linked list-based Implementation of Stack



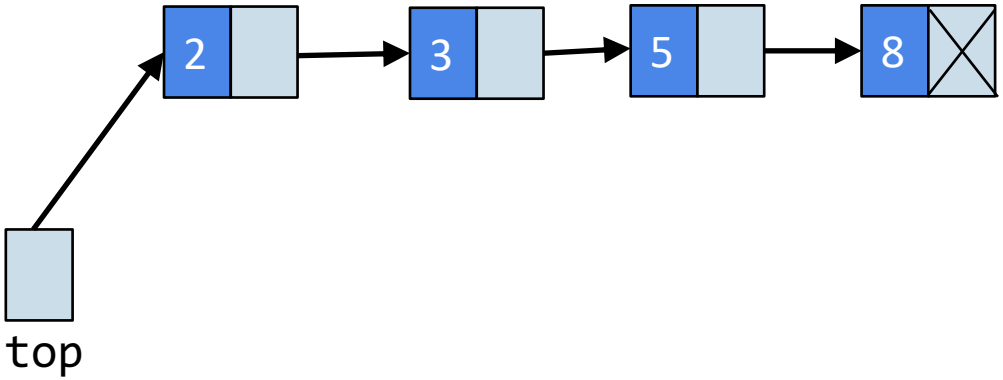


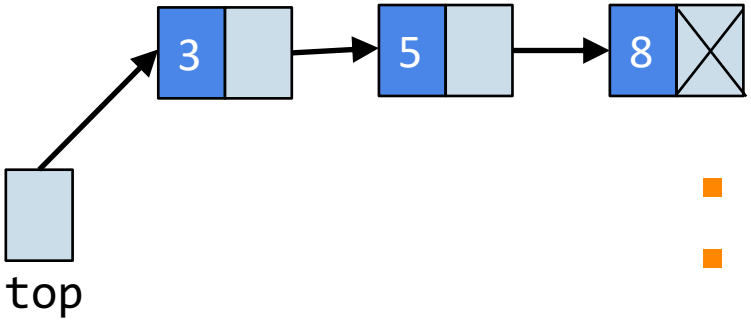
After the operation

```
push(&stack, 2);
```

- Insert element at the beginning
- `addElementFirst(&stack, 2);`

Linked list-based Implementation of Stack



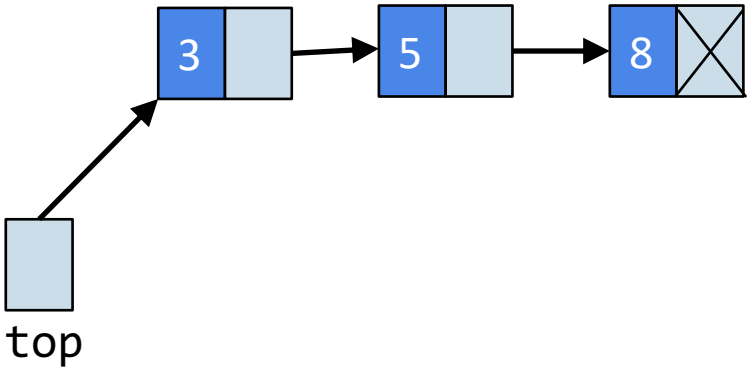


After the operation

```
pop(&stack);
```

- Remove the first item from the list
- `removeFirstElement(&stack);`

Linked list-based Implementation of Stack

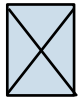


After the operation

```
peek(stack);
```

- Returns 3 (first element)

Linked list-based Implementation of Stack



- An empty stack == an empty list

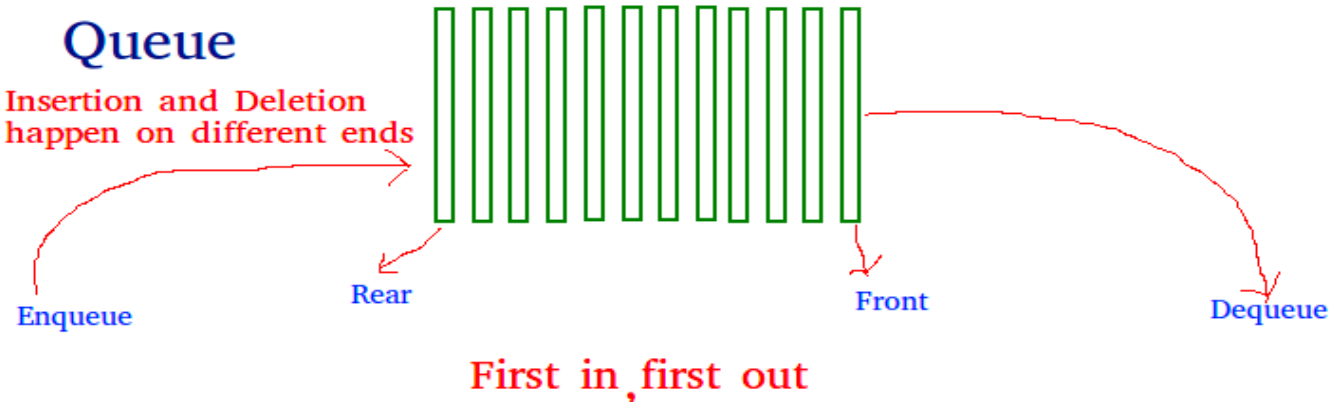
- A stack implemented as a linked list is full when there is not enough memory

Queue

- Queue is an abstract data type/linear data structure
- A queue has two ends
  - ✓ one end is used to insert data,
  - ✓ the other end is used to remove data from the queue

- It works like the line in front to the cash machine or the check-in at the airport:
  - ✓ You can get in line (enqueue). . .
  - ✓ When is your turn, you are in front. . .
  - ✓ Then, when you are done, you get out of line (dequeue).

Queue



Items are processed in the order that they arrive. This order is called FIFO (First-In, First-Out).

Image source: https://www.geeksforgeeks.org/queue-data-structure/ (used only for educational purpose)

Queue Operation

```
/* Creates an empty queue */
Queue createQueue(void);
```

Precondition:

- None

Postcondition:

- The returned value is an empty queue

Queue Operation

```
/* Add to queue */
void enqueue(Queue* queue, const Data data);
```

Precondition:

- Queue is not full
  - Theoretically, a queue can not be full. But in practice there are limitations depending on the implementation.

Postcondition:

- `data` is the last element in the `queue`

Queue Operation

```
/*Remove from Queue*/

void dequeue(Queue* queue);
```

Precondition:

- Queue is not empty

Postcondition:

- `Queue` contains one less elements after the operation

Queue Operation

```
/* Returns the first item in the queue */
Data front(const Queue queue);
```

Precondition:

- Queue is not empty

Postcondition:

- None

This operation is called peekQueue in the lab skeleton.

Queue Operation

```
/* Returns 1 if queue is full, 0 otherwise */
int isFull(const Queue queue);
```

Precondition:

- None

Postcondition:

- None

Queue Operation

```
/*Returns 1 if queue is empty, 0 otherwise */
int isEmpty(const Queue queue);
```

Precondition:

- None

Postcondition:

- None

Queue Interface/behavior

```
Queue createQueue(void);
void enqueue(Queue* queue, const Data data);
void dequeue(Queue* queue);
Data front(const Queue queue);
int isEmpty(const Queue queue);
int isFull(const Queue queue);
```

Example : Queue Usage

```
#include "queue.h"

int main(void)
{
   Queue queue = createQueue(); /* Creates a new empty queue */
   assert(isEmpty(queue)); /* Make sure the queue is really empty */
   enqueue(&queue, 5); /* Add 5 to the queue */
   enqueue(&queue, 8); /* Add 8 to the queue */
   assert(front(queue) == 5); /* Since 5 were added first, it's first
                                 in the queue */
   dequeue(&queue); /* Removes the first item from the queue(5)*/
   assert(front(queue) == 8); /*Since 5 is removed, 8 should be at the
                                front of the queue */
   dequeue(&queue); /*Queue should be empty now*/
   assert(isEmpty(queue));
   return 0;
}
```

Exercise: Queue

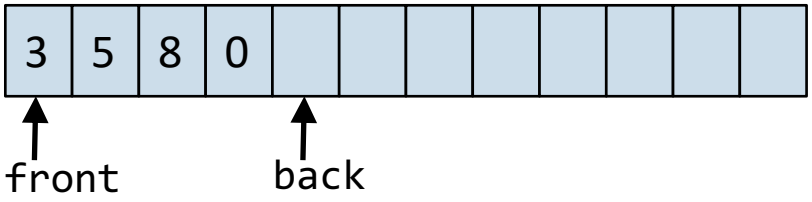
```
#include "queue.h"

int main(void)
{
    Queue queue = createQueue();
    enqueue(&queue, 3);
    enqueue(&queue, 7);
    enqueue(&queue, 1);
    dequeue(&queue);
    enqueue(&queue, 5);
    dequeue(&queue);
    dequeue(&queue);
    enqueue(&queue, 2);

    return 0;
}
```

What does the queue contains before returning from the main function?

Implementation of Queue

- We have now looked at the Queue interface.
  - Note that we do not know yet how Queue is implemented
  - The test program only considers that there is an implementation and that the queue data type is given.
- A queue can be implemented (just like stack) using different data structures, E.g.,
  - Array-based implementation
  - Linked-list based implementation

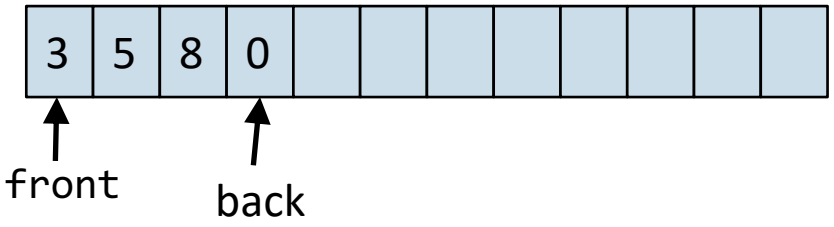
Array Implementation of Queue

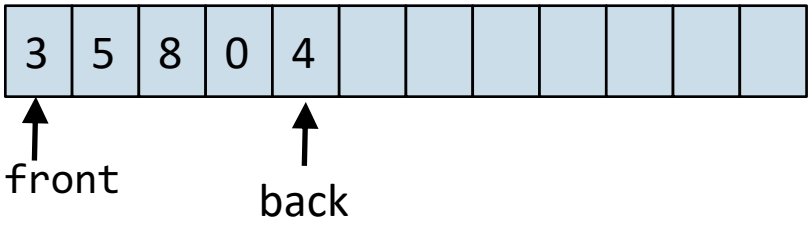



```
Struct QueueArray
{
    Data arr[MAXLENGTH];
    int front;
    int back;
};
```

- The `QueueArray` structure has the following items
  - `arr` to hold the data item
  - `front` index from where data can be removed
  - `back` index where data can be inserted
- **typedef struct** `queuearray` Queue;
  Queue now refers to a data type
- Initialize front=-1, back=-1

Array Implementation of Queue





Contents of queue after inserting
3 5 8 0 4

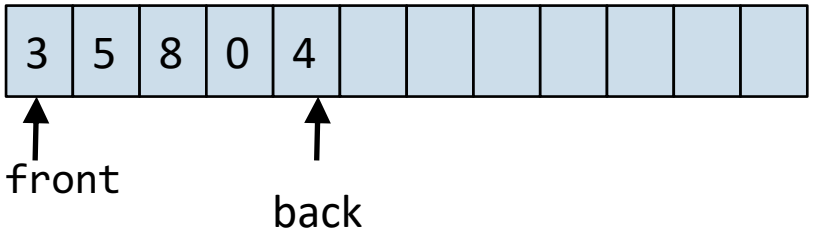
Operation Enqueue:

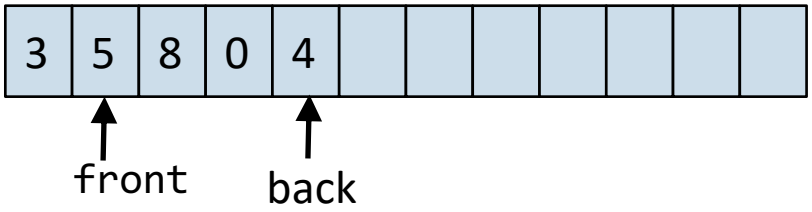
```
enqueue(&queue,4);
```

Algorithm:

1. Check that `queue` is not full
1. forward "back"
2. Insert item in the "back".
3. `if front=-1, set front=0`

Array Implementation of Queue





Contents of queue 5 8 0 4

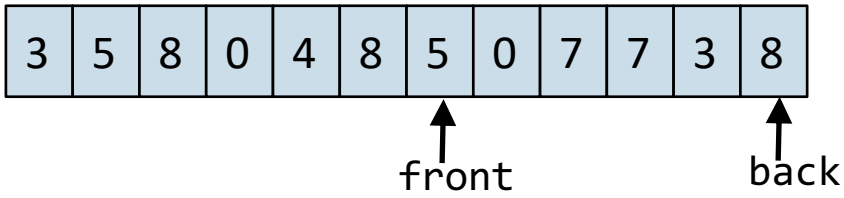
Operation Dequeue:

```
dequeue(&queue);
```

Algorithm:

1. Check `queue` not empty
2. Advance `front`

Array Implementation of Queue

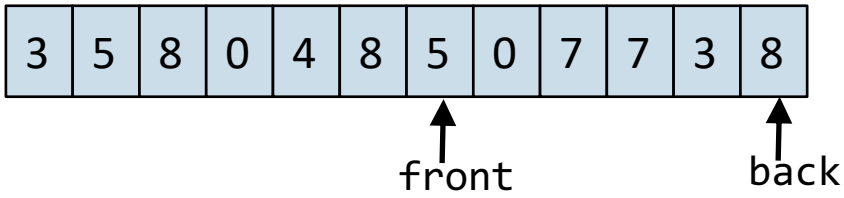


Contents of queue
5 0 7 7 3 8

After we added and removed some items

Is `queue` full?

Array Implementation of Queue



Contents of queue
5 0 7 7 3 8

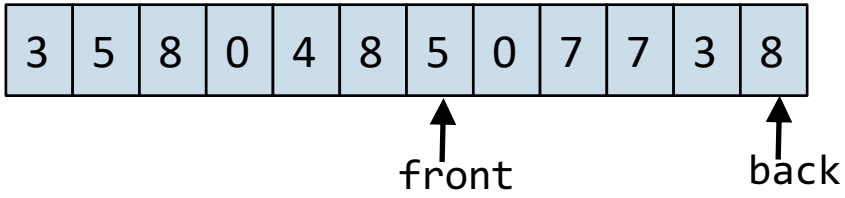
After we added and removed some items

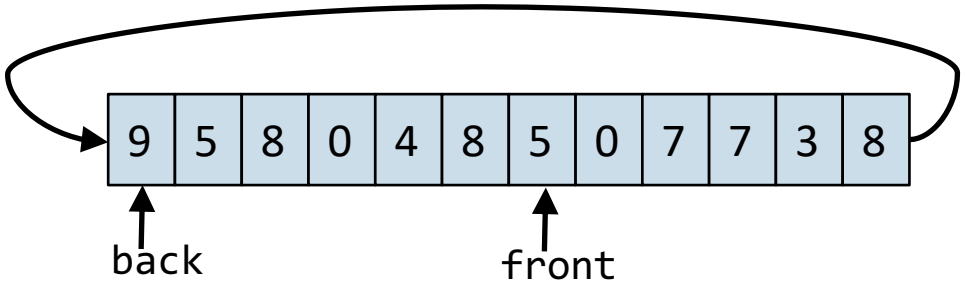
Is `queue` full?

Array has 12 places, but contains only 6 elements

`queue` is full if back ==MAXLENGTH-1

Circular Queue

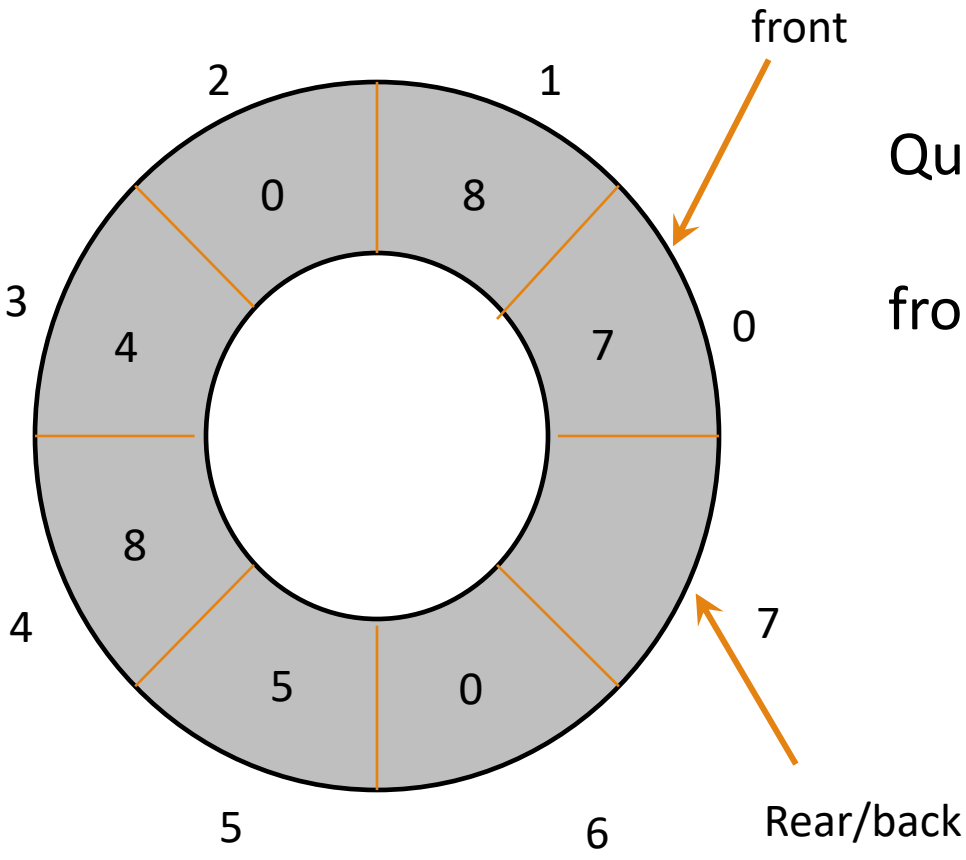




If `back` can move from the end to the beginning of the array, then `queue` is circular.

`queue` still contains elements between `front` and `back`, but can jump back when we reach the end of the array.

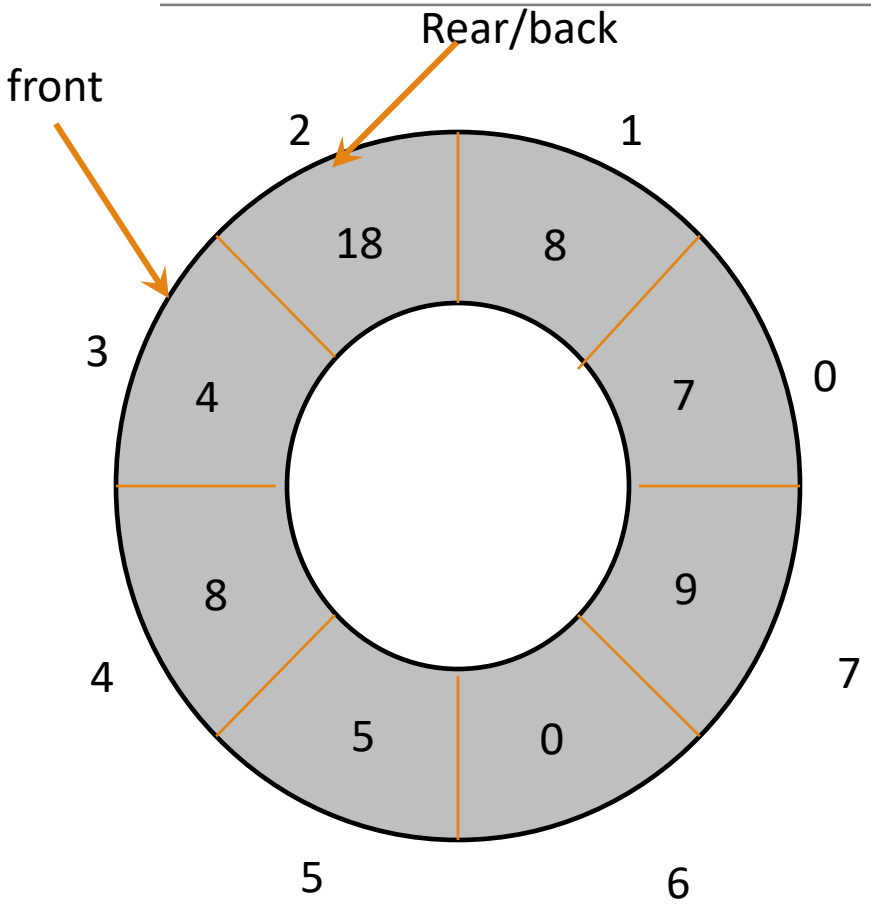
Circular Queue



Queue is full when

front=0 and back=MAXLENGTH-1

Circular Queue



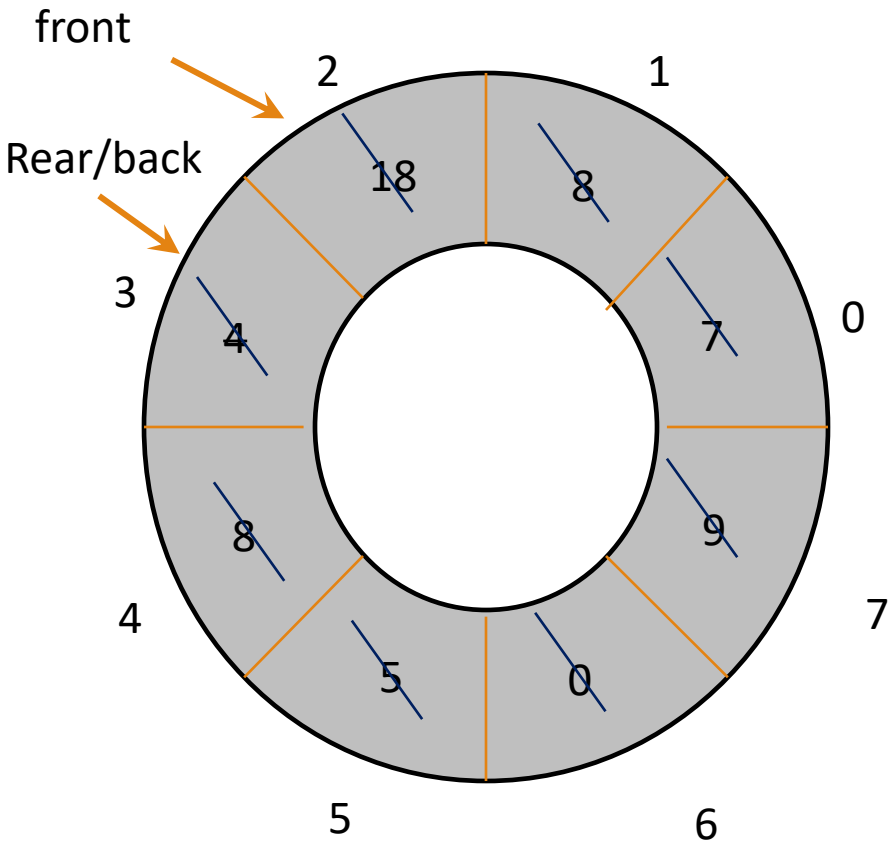
Queue is full when

front=0 and back=MAXLENGTH-1

Or

front=back+1

Circular Queue



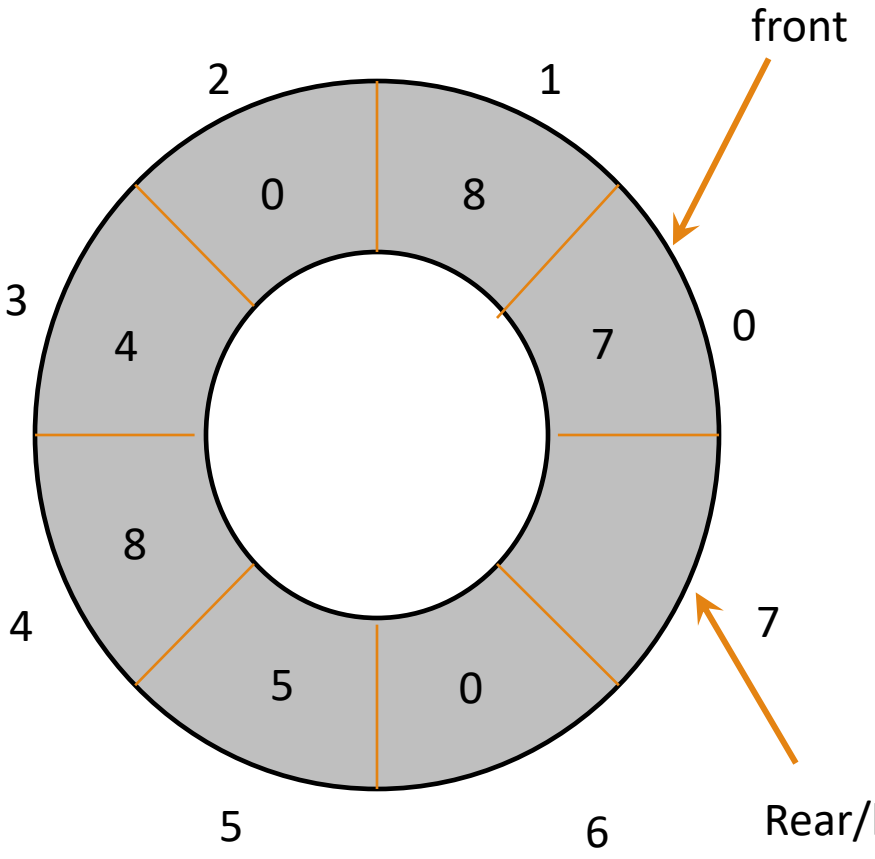
Queue is `empty` if

Rear=front and front=-1

When front=back and front!=-1:
Queue has only one element
After removing last element, set

front=-1, rear=-1

Circular Queue

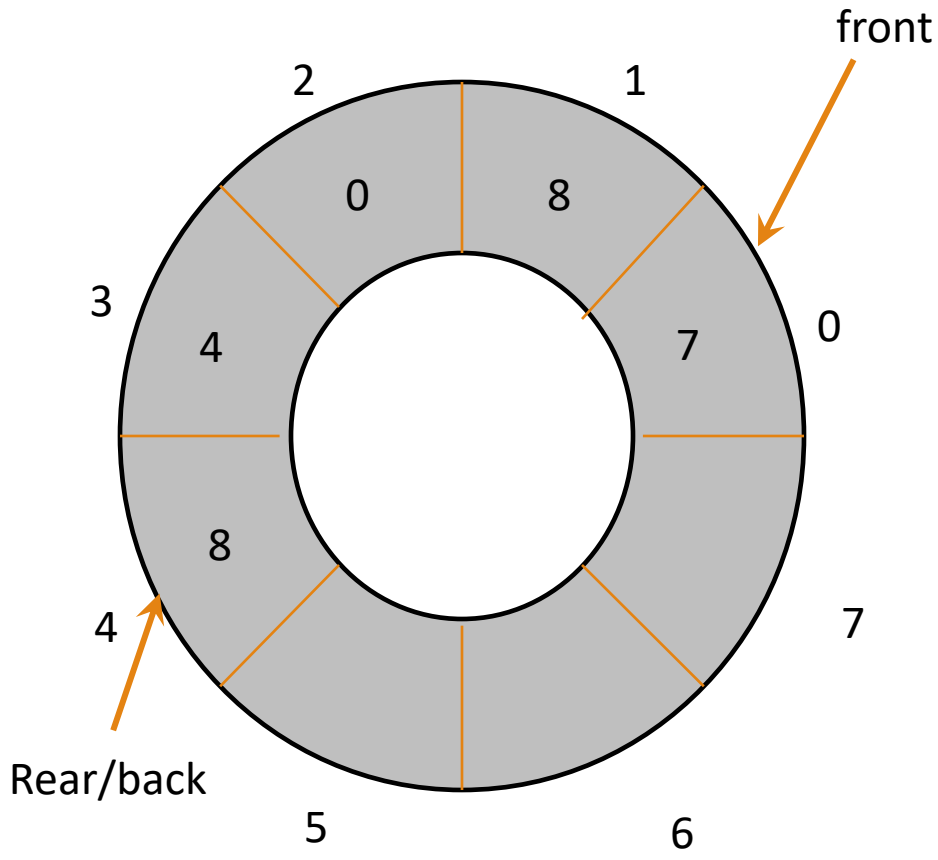


Algorithm for `enque` operation:

1. Check that `queue` is not full
1. forward "`back`"
2. Insert item in the "`back`".
3. `if front=-1, set front=0`

same algorithm like non-circular queue

Circular Queue

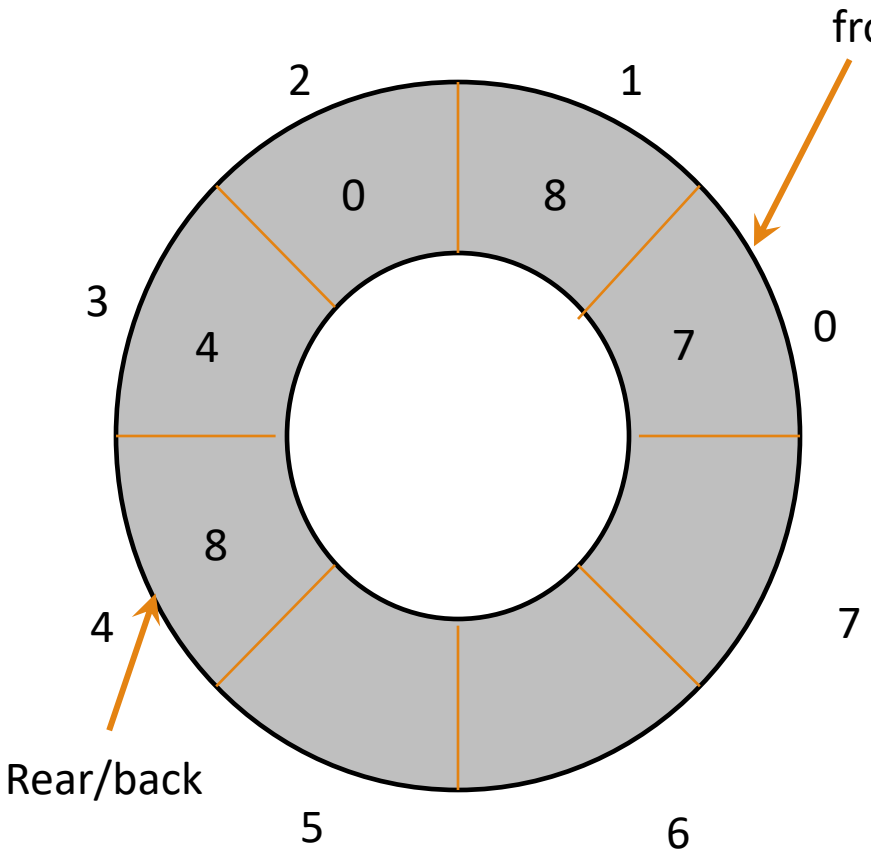


Algorithm for `enque` operation:

1. Check that `queue` is not full
1. forward "back"
2. Insert item in the "back".
3. `if front=-1, set front=0`

same algorithm like non-circular queue

Circular Queue

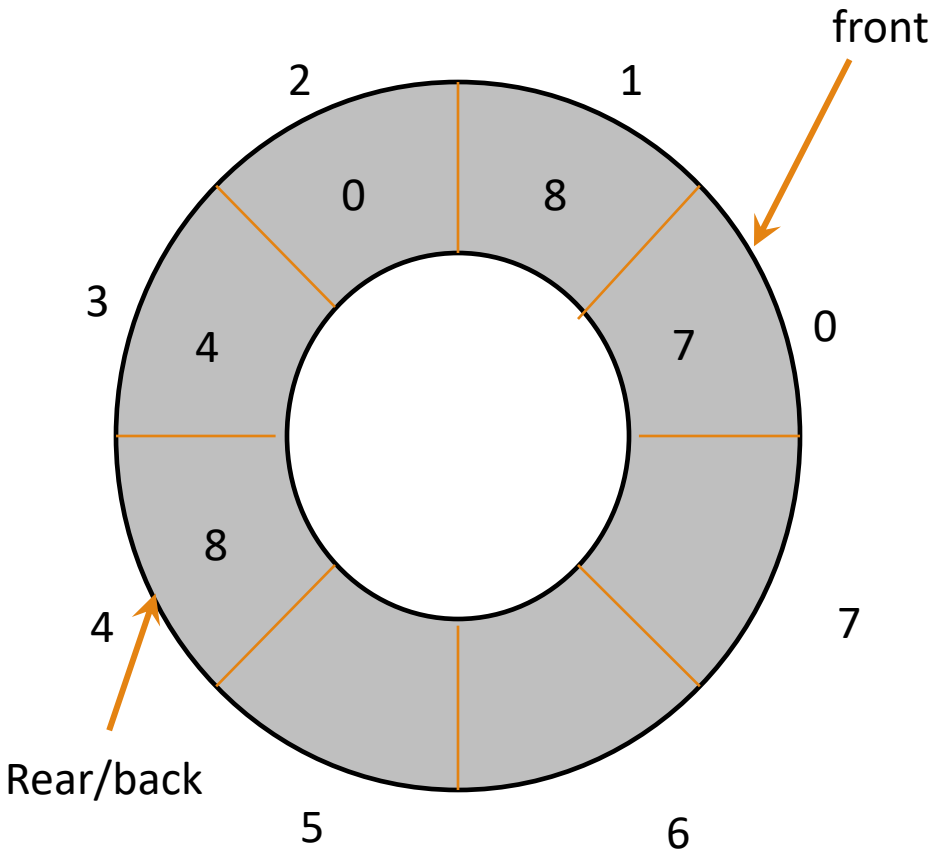


So, what's the trick to forward back?

```
if (front == -1)
      back = 0;
 if (back == MAXLENGTH-1 && front != 0)
      back = 0;
  else
      back++;
```

back = (back + 1) % MAXLENGTH

Circular Queue



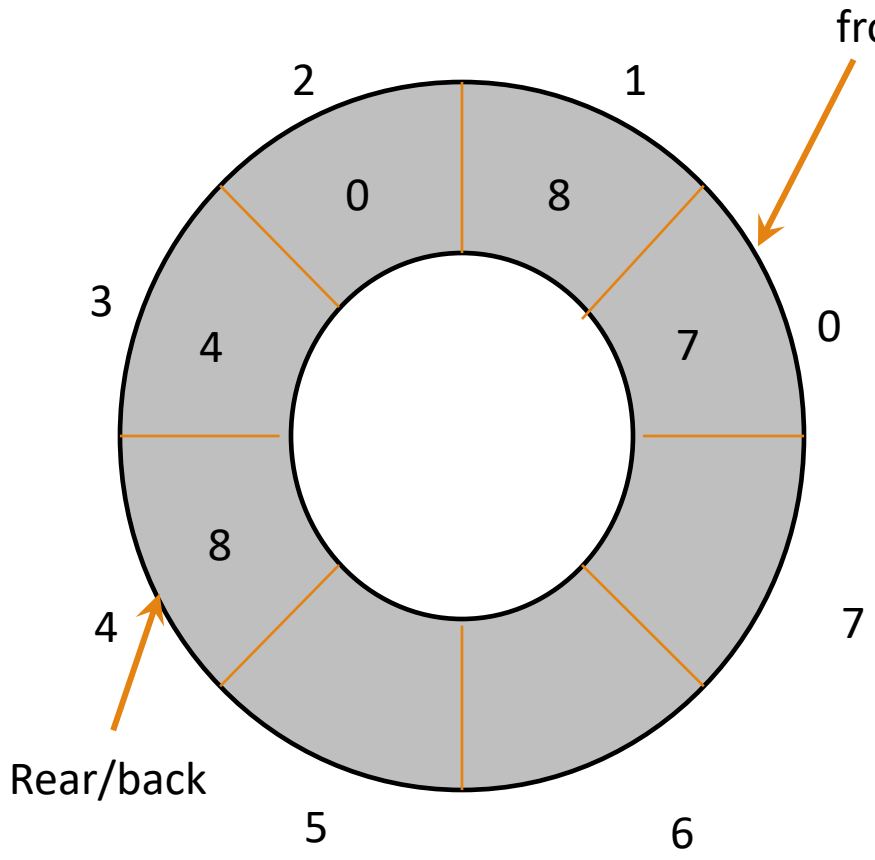
Algorithm for `deque` operation:

Algorithm:

1. Check `queue` not empty
2. Advance `front`

same algorithm like non-circular queue

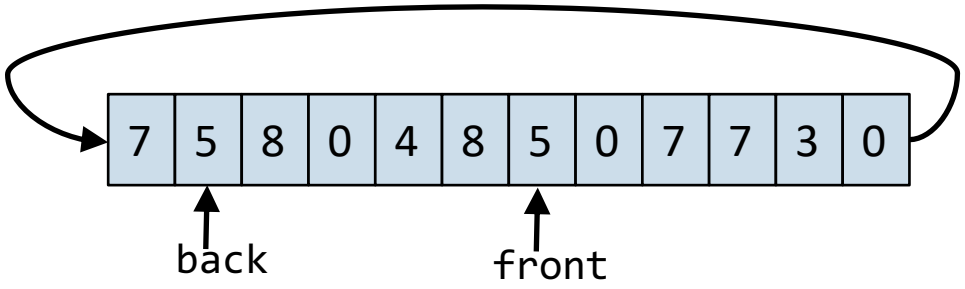
Circular Queue

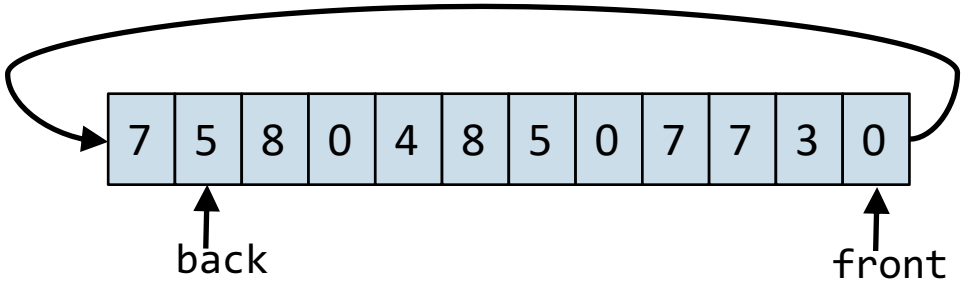


So, what's the trick to forward front?

You shall find it out (similar to back)

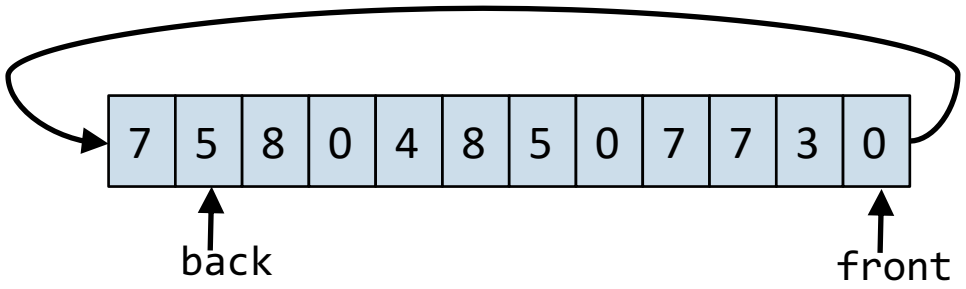
Example: Circular Queue

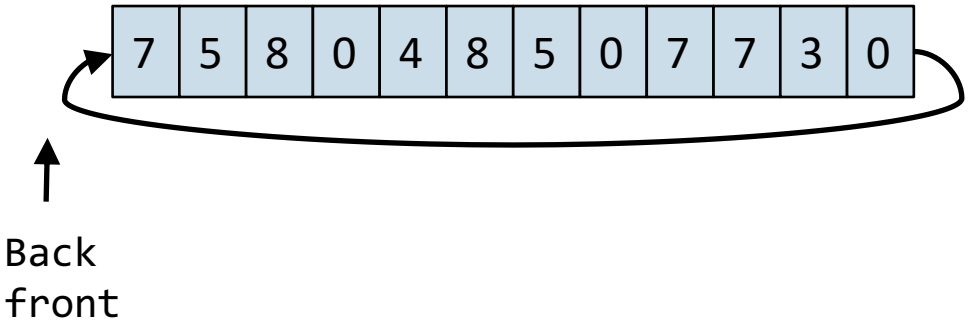





```
for(i=0; i < 5; i++)
    dequeue(&queue);
```

Example: Circular Queue





After the Operations:

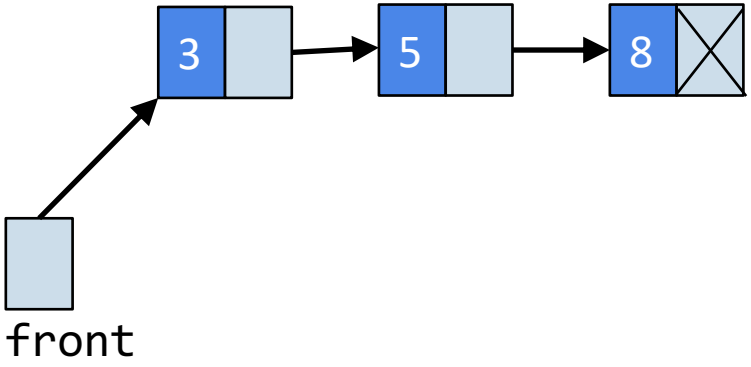
```
dequeue(&queue);
dequeue(&queue);
dequeue(&queue);
```

Advance `front` 3 times

!!! Remember: when front==back and front!=-1 and `dequeue(&queue);`

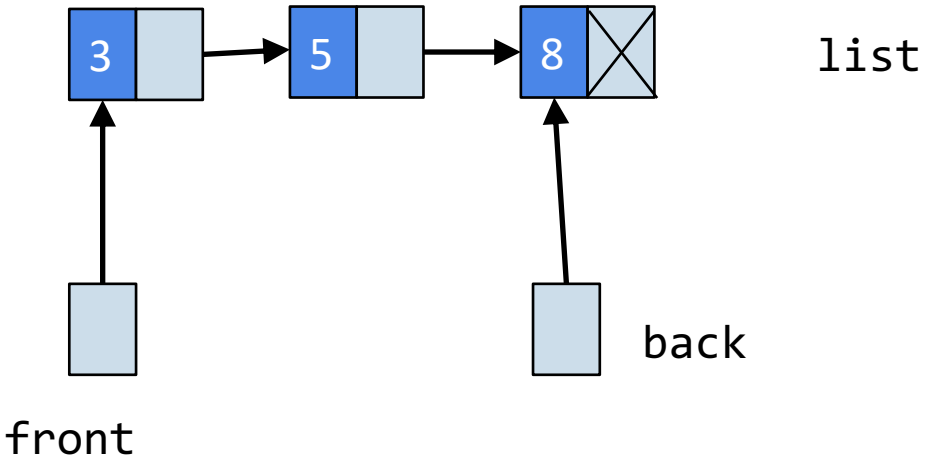
`Set front=-1, back=-1`

Linked list-based Queue Implementation



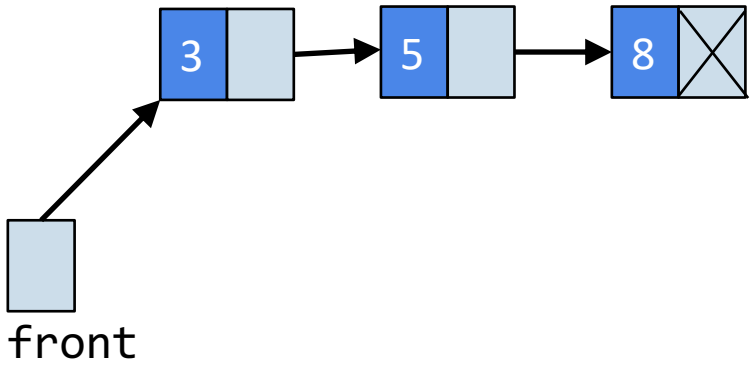
- We implement Queue as linked-list by keeping two pointers pointing to the `front` and `rear/back` element.
- We can declare Queue type as:
  - `typedef List Queue;`

Linked list-based Queue Implementation



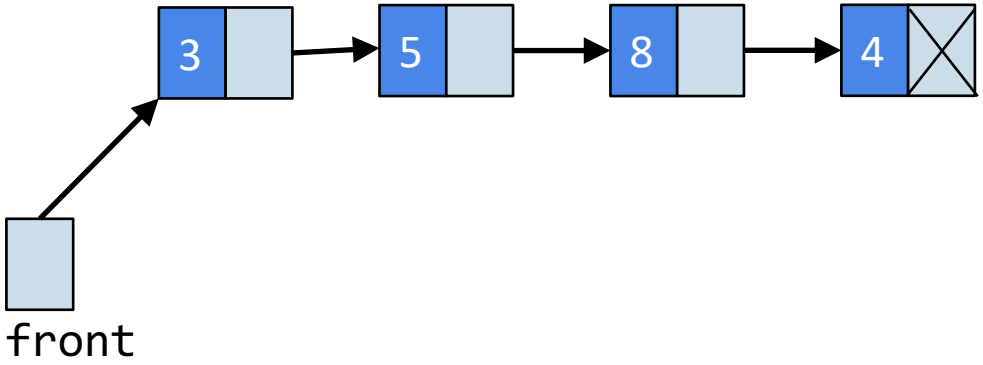
- Remember the `head` pointer we used in linked-list
- For queue, now we have `front` and `back` pointer…

Linked list-based Queue Implementation



Operation:

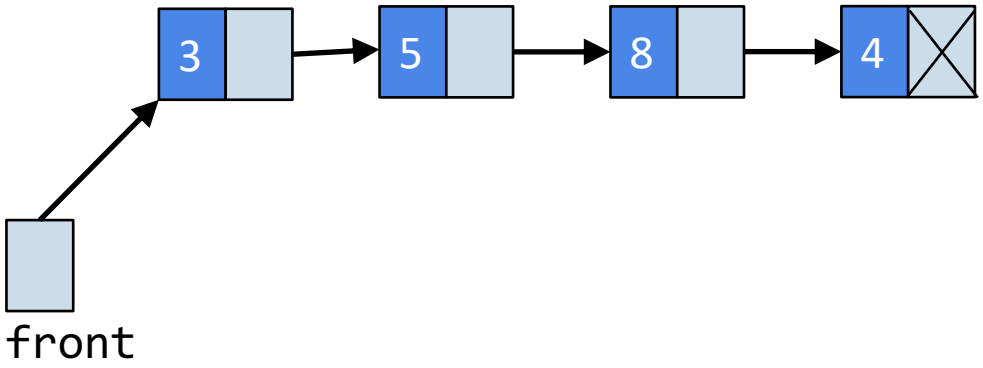
```
enqueue(&queue, 4);
```

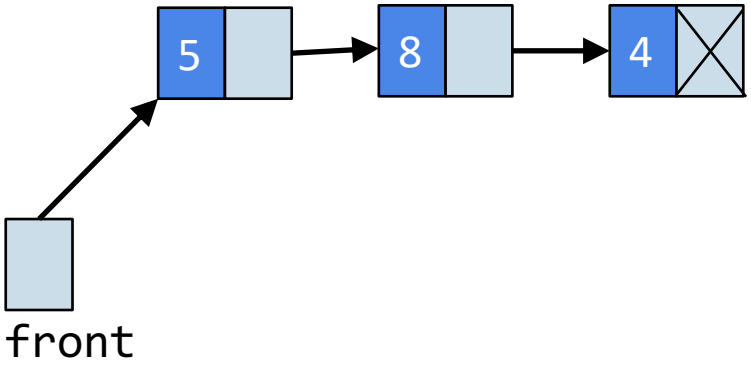


Algoritm:

1. Create a new node
2. Add the node to the back of the list
3. No need to search the list (use the back pointer)

Linked list-based Queue Implementation





Operation:

`dequeue(&queue);`

Algoritm:

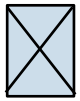
1. Remove the element pointed to by the front pointer (if front!=NULL)

Operation

`front(queue);`

Returns 5 after the `dequeue`

Linked list-based Queue Implementation



A queue is `empty` if the list is `empty`

A queue implemented as a linked list can not be full. However, it is possible that there is not enough memory to allocate, and you cannot perform the `enqueue` operation.

Exercise

Assume an array implementation of a queue (circular), the queue is initially empty and has 7 places (7 elements). Of course, we have a front and a back.

We execute the followings:

```
enqueue(1); enqueue(2); enqueue(3);
dequeue(); enqueue(4); dequeue();
enqueue(5); dequeue(); dequeue();
enqueue(6); enqueue(7); enqueue(8);
enqueue(9); enqueue(10); enqueue(11);
dequeue()
```

Is there any problem?

Set

In mathematics, a `set` is a collection of distinct objects which are called member or elements of that `set`.

Example:

I={2,9,0,-1} // a set of integer numbers

C={blue, yellow, red} // a set of colors

S={ABBA, Agnes, Alcazar }. // a set of ?

The elements are distinct:

{2,2,3,4,4,4,5} = {2,3,4,5}

There is no particular order on the elements of set:

{2,3,4} = {3,2,4}

Set

We perform certain operations on sets.

- Check if an element belongs to a set (membership check):

$$2 \in \{2,3,4\}, \text{ but } 5 \notin \{2,3,4\}$$

- The union of two sets:

$$\{2,3\} \cup \{3,4\} = \{2,3,4\}$$

- The intersection of two sets.

$$\{2,3\} \cap \{3,4\} = \{3\}$$

- Subset (the elements of one set contains in another set)

$$\{2,3\} \subset \{2,3,4\} \text{ and } \{2,3\} \subseteq \{2,3\}$$

ADTn Set

In computer science, `Set` **is an** `abstract data type` **that can store unique values, without any particular order.**

We implement the `mathematical concepts of finite sets` by means of ADT.

ADTn Set

Set usage examples:

```
addElement(set,5) /*set contains element 5*/
addElement(set,8) /*set includes elements 5
 and 8*/
isInSet(set,3) /* checks if 3 is in the set.
                Answer: no*/
inInSet(set,8) /* checks if 3 is in the set.
                Answer: no */
addElement(set,8) /* 8 already exists in the
 set. set
            does not change*/
```

Set Application

- Set can be used to keep track of which elements have been processed.
- You can use the set to avoid duplicate letters.
  - If we have a Set of All Student IDs of MDH, we can check if an ID has been taken or not

Set Operations

```
/* Create an empty set */
Set createSet(void);
```

Preconditions:

- None

Postconditions:

- Returns an empty set

Set Operations

```
/* Add data to the set pointed to by the set
 pointer, if data pre-exists in the set, then
 nothing happens*/
void addElement(Set* set, const Data data);
```

Precondition:

- Set is not full (remember we consider finite sets in computer implementation)

Postcondition:

- `data` is now an element of the set `*set`

Set Operations

```
/* Removing data from the set *set */
void removeElement(Set* set, const Data
 data);
```

Pre-conditions:

- None
  - Set can be empty and nothing should happen then.

Post-conditions:

- `data` does not exists in the set `*set`

Set Operations

```
/* Returns 1 if data is in the set *set, 0
 otherwise */
int isInSet(const Set set, const Data data);
```

Precondition:

- None

Postcondition:

- None

Set Operations

```
/* Returns 1 if set is full, 0 otherwise */
int isFull(const Set set);
```

Precondition:

- None

Postcondition:

- None

ADTn Set: Interface

```
Set createSet(void);

void addElement(Set* set, const Data data);

void removeElement(Set* set, const Data data);

int isInSet(const Set set, const Data data);

int isFull(const Set set);
```

Implementations of the Set ADT

- We have now looked at the interface for a Set. We do not yet know how our Set is implemented - not even what data type / structure it is.
- The test program and how it works is completely independent of implementation
- A set can be implemented by means of:
  - An array
  - a linked list
  - A set can also be implemented by using a binary search tree (as we will see next week)

Implementations of the Set ADT

Like other ADTs, Set can be implemented as an array or a linked list. Because the position of the element is irrelevant so we do not need to visualize these.

Since the main task of a Set is to search for data (determine if a data belongs to a set or not), implementing set using binary search tree is more efficient.

Implementations of the Set ADT

```
void addElement(Set* set, const Data data);
```

Algorithm:

1. Check if data is already in the list / array
2. If yes, do nothing
3. If no, add data in the set (list / array / tree). Position is not important.

Implementations of the Set ADT

```
void removeElement(Set* set, const Data data);
```

Algorithm:

1. Check if data exists in the structure (list / array / tree)
2. If yes, then delete data
3. If no, do nothing.

Implementations of the Set ADT

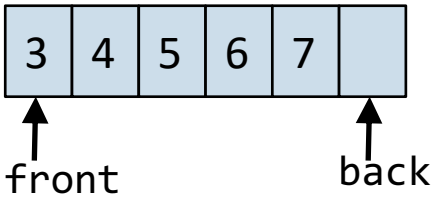
```
int isInSet(const Set set, const Data data);
```

Algorithm:

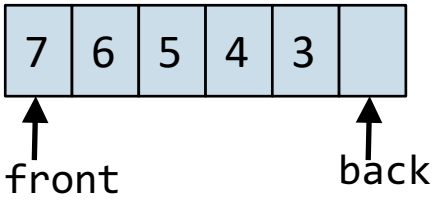
- search the list / array / tree, return 1 if data is found and 0 otherwise.

Exercise

Suppose we have the following queue



We want to reverse the order so that it looks like this:



How can we solve this?

Graph

Graphs can be used to model a variety of types of problems

- Road network / Street network
- Communication network
- Flight Routes
- Program structure and program flow
- computer Games (to model possible moves)
- etc….

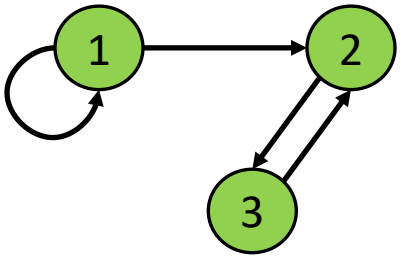
Graph

- Graph can be used to represent a real life problem. Then we need to find solutions of the problem by performing various operations on the graph.
  - Structure data → easier to find subproblems that need to be resolved
  - Shows on relationships (facebook friendlist)
  - Easy to visualize the problem → easier to find a solution

Graph

Definition: A graph *G* is a pair *(V, E)* *(i.e. G=(V,E))* such that

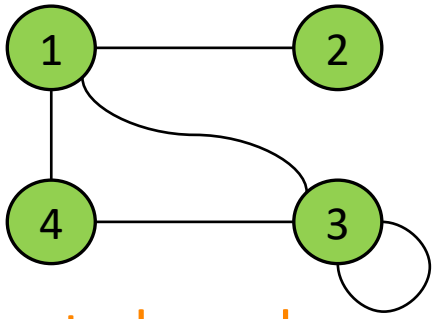
- V is a set of vertices (also called nodes) of the graph G
- E is a set containing the relation between vertices representing graph edges



Directed Graph

V = {1, 2, 3}

E = {(1,1), (1,2), (2,3), (3,2)}



Undirected graph

V = {1, 2, 3, 4}

E = {(1,2), (1,4), (1,3), (3,4), (3,3)}

Graph

- The nodes in a graph always contains some information depending on what the graph represents.

  For example, A graph representing a road network can have city names (or maybe pointer to more info).

- We now number them to easily manage them

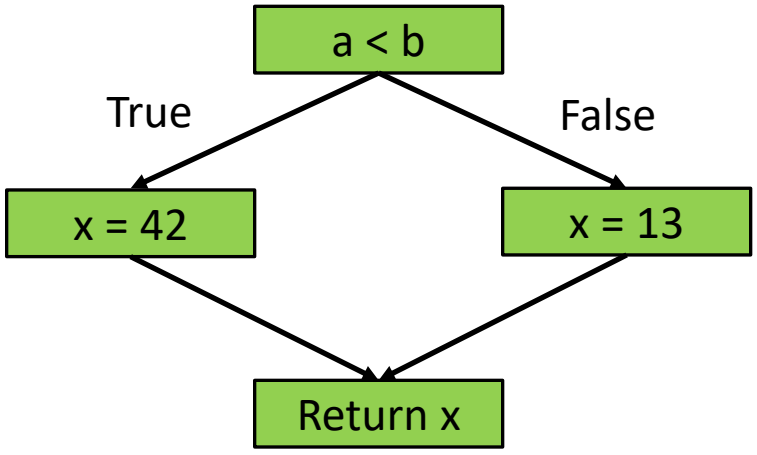
- Often, you only use numbers in the nodes, and allow these numbers to refer to the data objects (for example, as an index in an array)

Graph

The edges may contain information

The distance between:

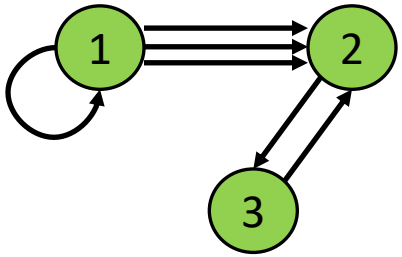
Visualization of program sections:



Graph

Graphs can be multigraphs

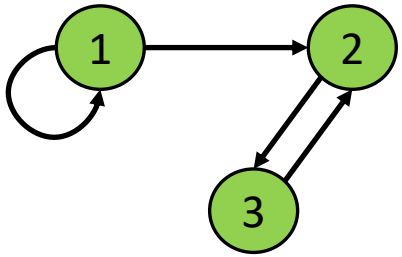
That is, there may be multiple arcs between pairs of nodes



Graph

A path in a graph is a list of nodes n_1, n_2, n_3, …, n_k such that (n_i, n_{i+1}) ∈ E for all I between 1 and k-1

The graph below contains, for example, the paths (1,2,3), (2,3,2) and (1,1,1)



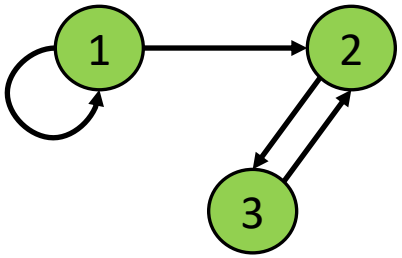
A path may contain a cycle (loop).

For example: (1, 1) and (2,3,2)

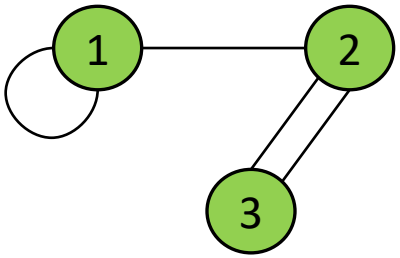
Graph

A node is said to be adjacent to another node if there is an edge from the second node to the first node

2 is adjacent to 1.



1 and 2 are adjacent to each other



In directed graphs, the term "successor" is often used instead

Graph

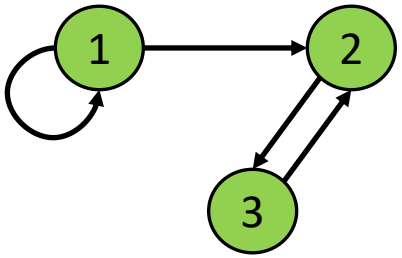
A graph is connected (sammanhängande) if there is a path between each pair of nodes.

A directed graph G is connected if the unidirectional graph formed by replacing all the directed edges in G with the undirected edges being connected

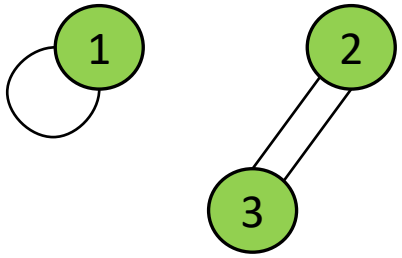
A graph that is not connected is disconnected

A directed graph where each node can be reached from any other node via the directed edges is called strongly connected.

Graph



Connected Graph



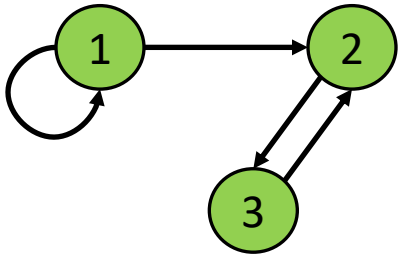
disconnected graph

Graph

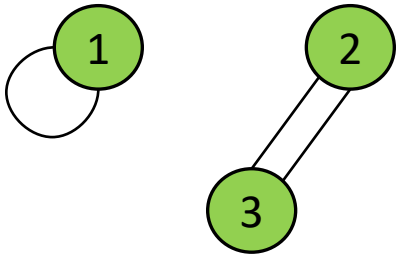
The degree of a node indicates the number of incoming edges

In directed graphs, we distinguish between incoming and outgoing edges

Ingrad(1) = 1 utgrad(1) = 2
Ingrad(2) = 2. utgrad(2) = 1
Ingrad(3) = 1 utgrad(3) = 1

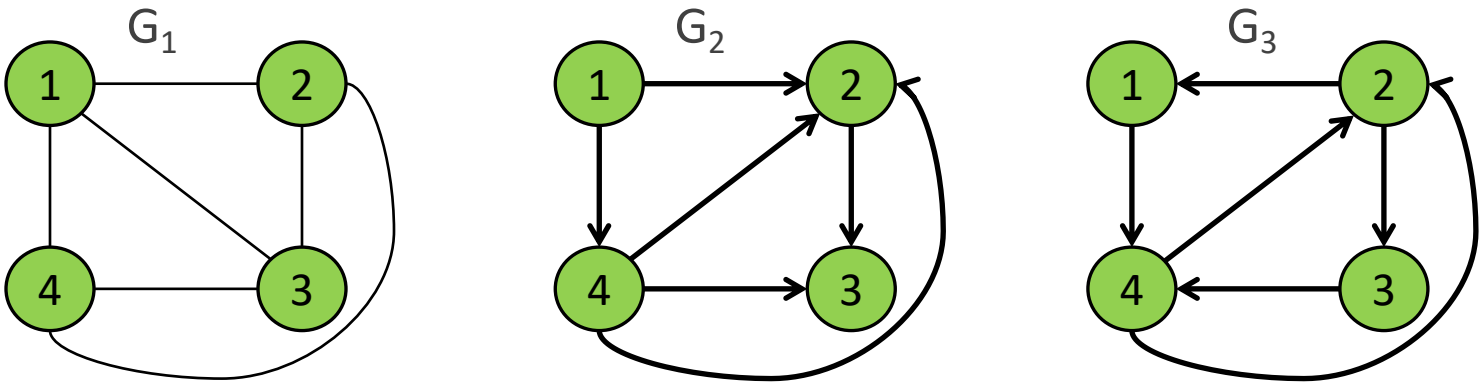


Grad(1) = 2
Grad(2) = 2
Grad(3) = 2



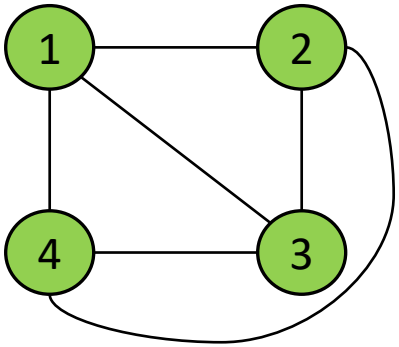
Graph - Exercise

- Which graphs are connected?
- Which contains cycles, and what are the cycles?
- Which are multigraphs?
- What degree does Node 2 have in each graph?



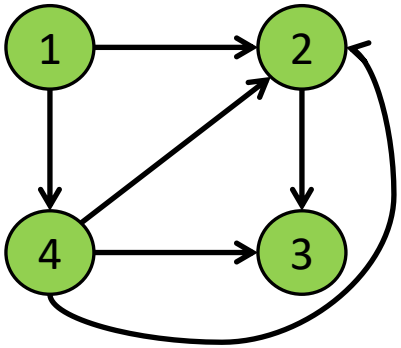
Solution

G_1



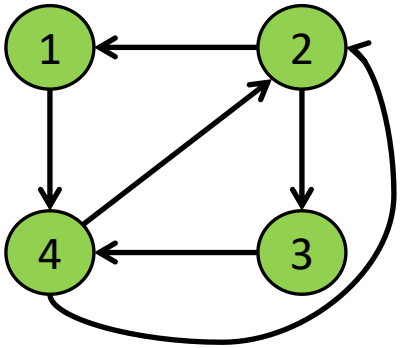
- Connected
- Cycle(ex. 1,2,4,1)
- Not multigraph
- Grad(2) = 3

G_2



- Connected
- No cycle
- Multigraph: *multiple arcs between (4,2)*
- Ingrad(2) = 3
- Utgrad(2) = 1

G_3



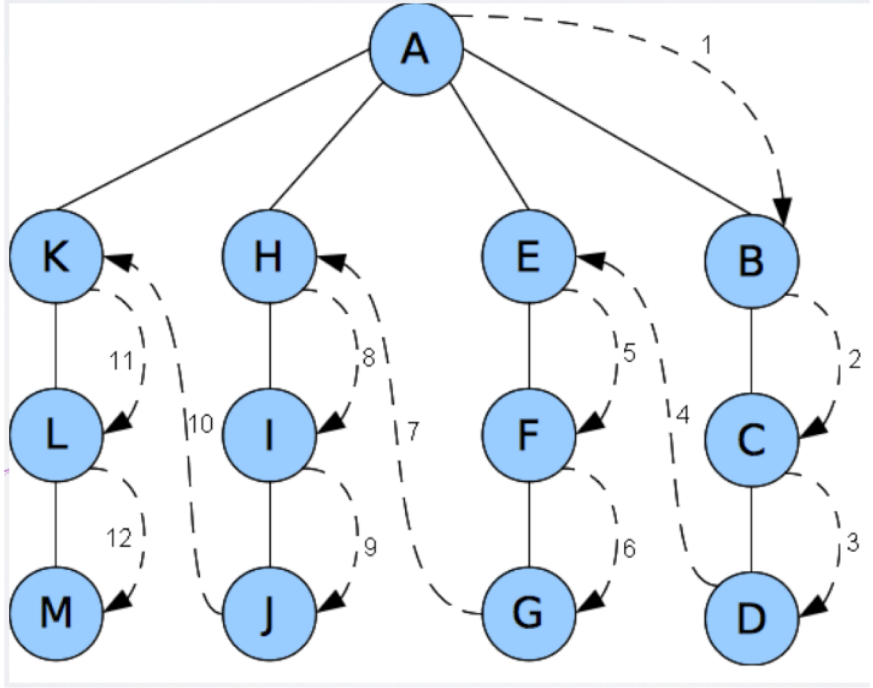
- Strongly connected
- Cycle (Ex. 1,4,2,1)
- Multigraph *multiple arcs between (4,2)*
- Ingrad(2) = 2
- Utgrad(2) = 2

Traversing the Graph

- In which order do we visit the nodes in a graph?
- In a connected graph, you can visit nodes according to the breadth or the depth.
- The traverse begins from a certain node
- Often there is a particular node in the problem that is of particular importance.
- For example. We may want to find a graph of the road network in a given location.
- Since a node may have multiple arcs, we need a flag to indicate if a node is visited or not?

Depth first traversal

Depth-first search (DFS) is a method for exploring the graph. In a DFS, you go as deep as possible down one path before backing up and trying a different one.

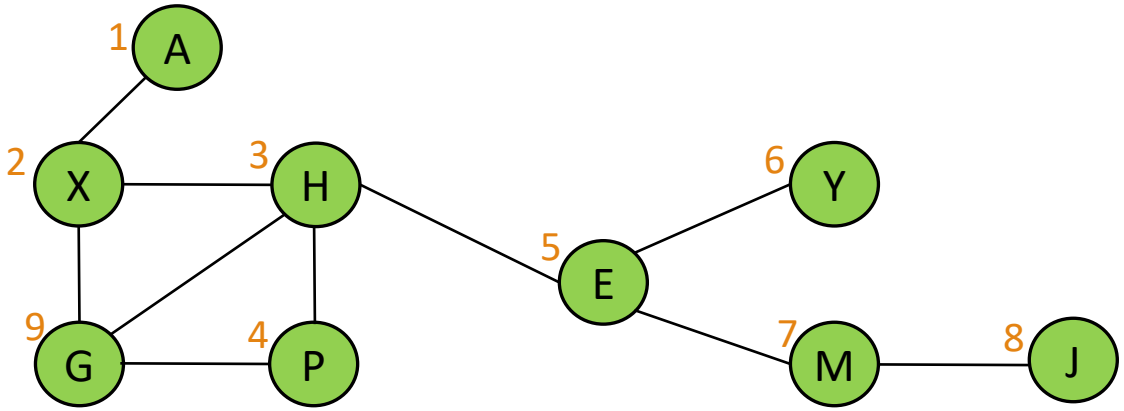


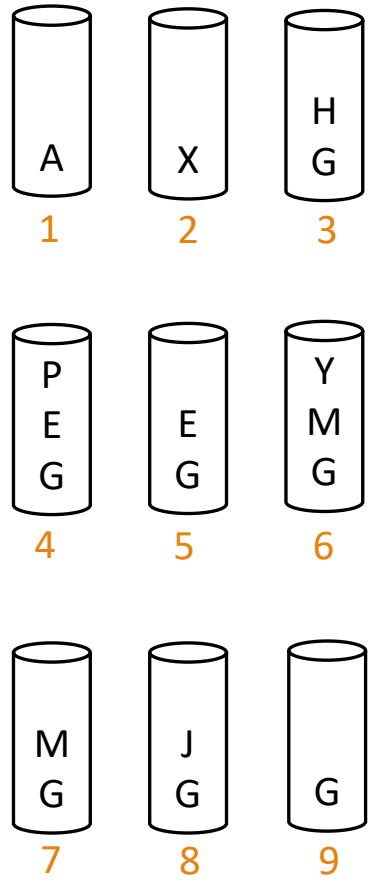
Depth first traversal

To perform the `Depth first search`, we need to use a stack

- Start by pushing the first node into the stack
- Looping
  - Pop node **`P`** from the stack
  - Process node **`P`**
  - Push all the nodes that are successor nodes of **`P`** and not put into the stack
- When the stack is empty, the traverse is complete

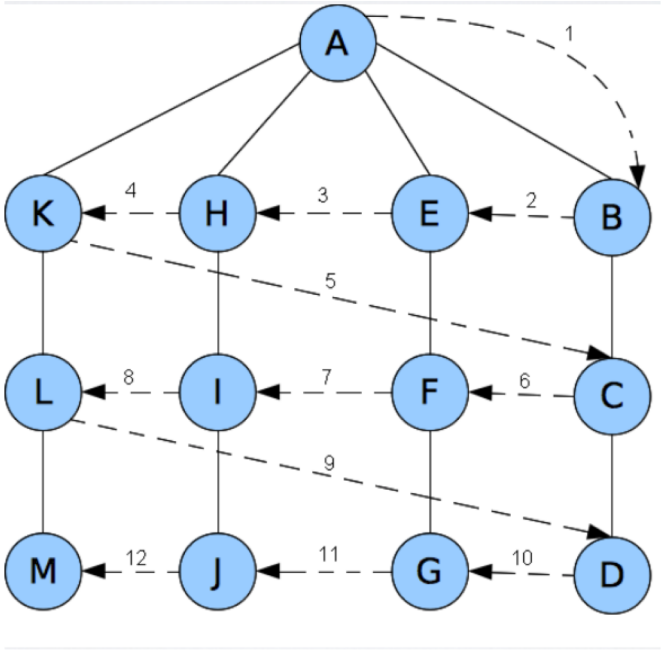
Depth first traversal





Breadth first traversal

We traverse through one entire level of successor nodes first, before moving on to traverse through the successors' sucessor nodes.

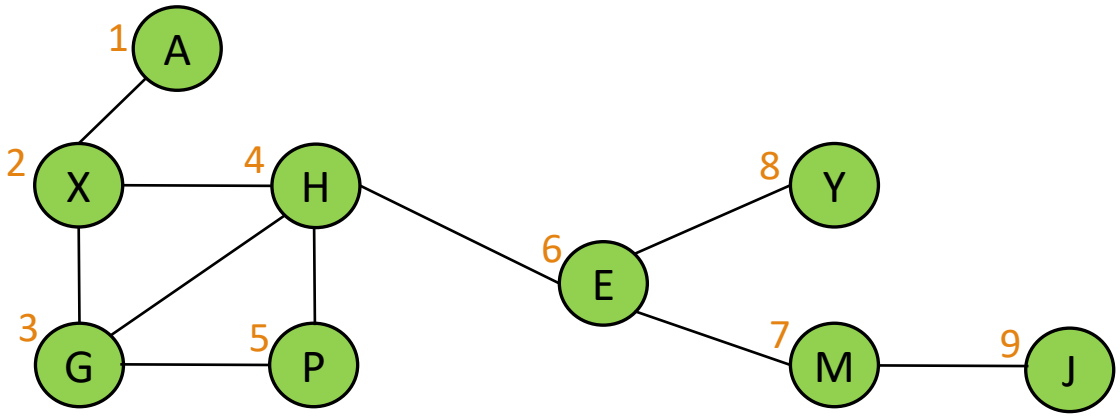


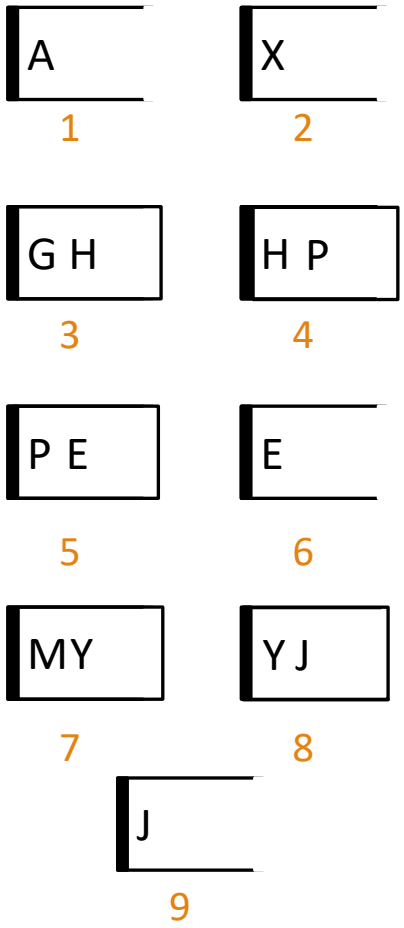
Breadth first traversal

To perform the breadth first search (BFS), we need to use a queue

- Start by adding the first node (enqueue)
- Looping
  - Remove first node `P` from the queue (dequeue)
  - Process that node
  - Add all the nodes that are successors of `P` (enqueue)
- When the queue is empty, the traverse is complete

Breadth first traversal





Interface

What do we want to do with a graph?

```
/* Create a new graph */
 Graph createNewGraph()

/* Adds a new node to the graph
   Pre: there is a graph
   Post: A new node with data is added */
 void insertVertex(Graph *graph, const void *key)

/* Removing a node - here you have to decide how / if you?
   handles a node with a degree > 0.
   Pre: There is a graph (which is not empty)
   Post: The node is deleted (as well as potential edges?
   To / from it - may need other nodes degrees here?))*/
int deleteVertex(Graph *graph, const void *key)
```

Interface (contd…)

```
/* create an edge between two nodes
   Pre: The graph exists (and is not empty)
        fromKey and toKey are pointers to nodes in the graph.
   Post: Edge is added (and the degrees are updated)
   Return: 1  - Successful
           -1 - fromKey is not available
           -2 - toKey is not available */
int addEdge(Graph *graph, const void *fromKey, const void
*toKey)

/* Remove an existing edge
   Pre: The graph exists (and is not empty)
        fromKey and toKey are pointers to nodes in the graph
   Post: The edge is deleted (and the degrees are updated)
   Return:  1 - successful removal
           -1 - fromKey is not available
           -2 - toKey is not available */
int deleteEdge(Graph *graph, const void *fromKey, const
void*toKey)
```

Interface (contd…)

```
/* Traverse the graph in different order (depth-first and
   breadth-first). For example, Print or search in
   the graph, if the operation is to search the graph, we need
   to have a return value

   Pre: There is a graph
   Post: Depends on what the operations should do
   Flag indicates whether the node has been processed or not, eg
        0 - not treated
        1 - is on the stack / in the queue
        2 - treated */

void/int depthFirstTraversal(const Graph *graph, void *flag,
                             (const void * keyTpFind))
void/int breadthFirstTraversal(const Graph *graph, void *flag,
                               (const void *keyToFind))
```

Representation

To represent a graph we need

- An array to store the nodes
- A matrix to the arcs

We can do this as well

- A one-dimensional array for the nodes
- A two-dimensional array for the edges

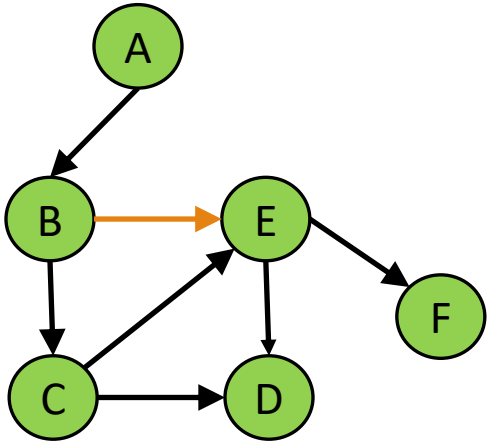
Or

- A "two-dimensional" linked list

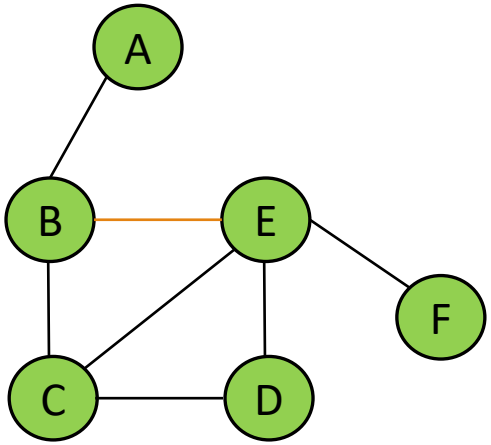
Representation (array)

A	B	C	D	E	F

Riktad matris



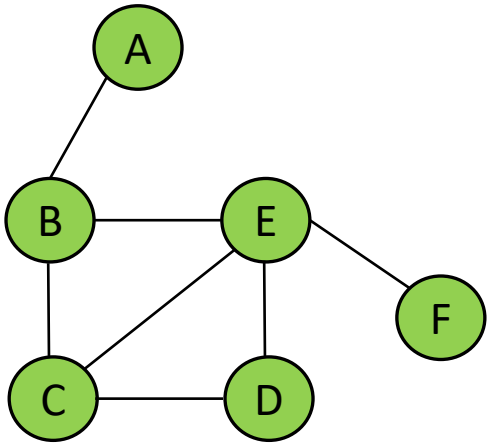
Oriktad matris

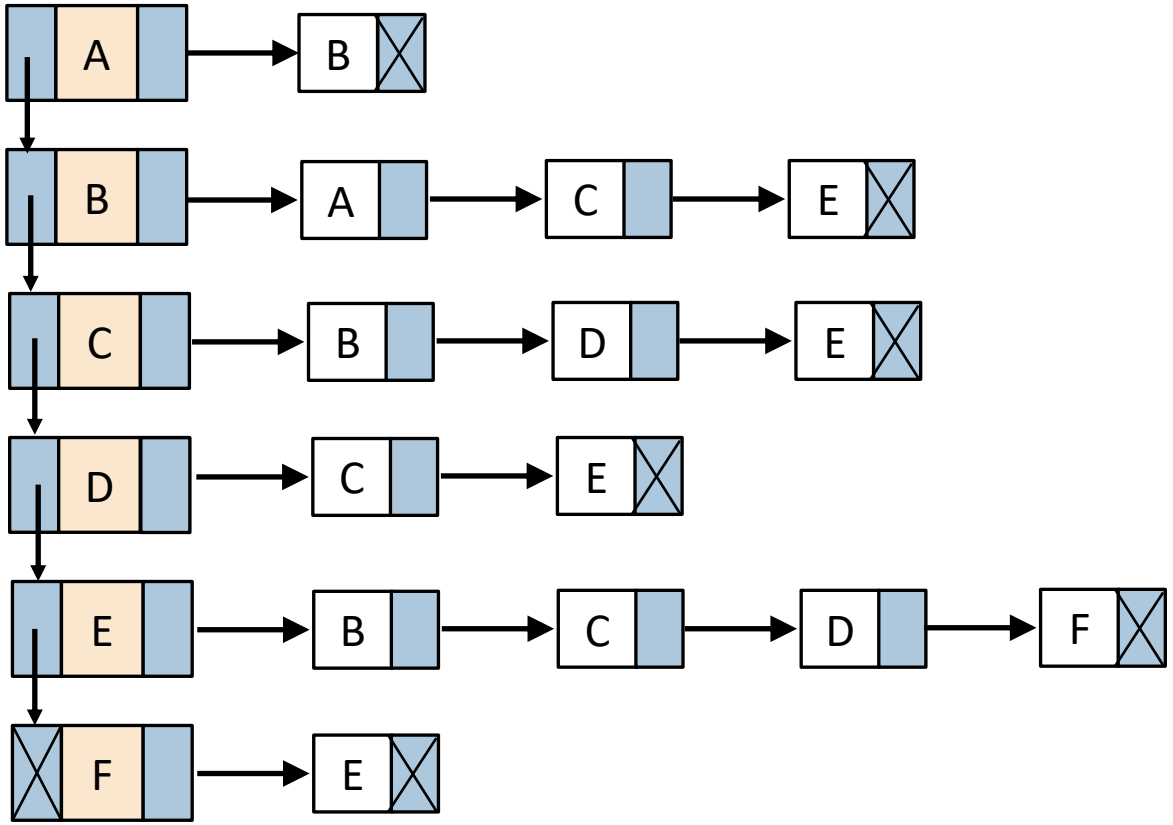


	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

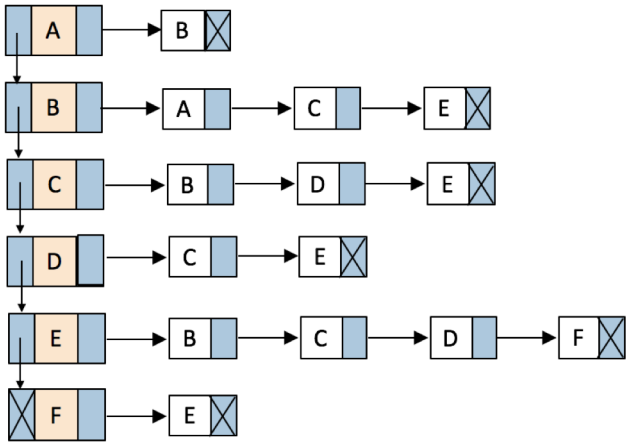
	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Representation (linked list)

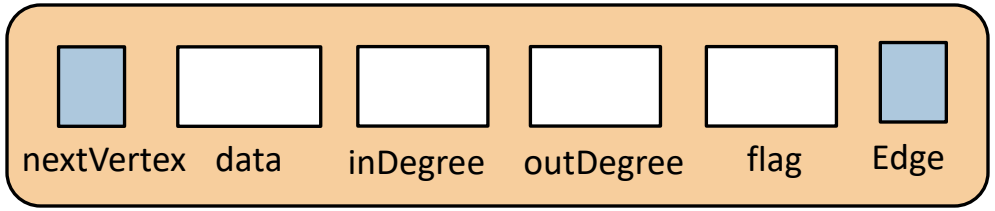




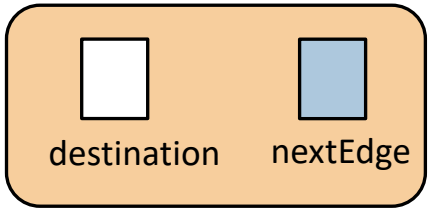
Representation (linked list)

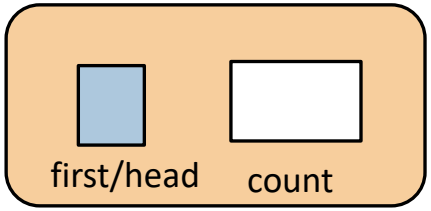


The node is implemented as a struct



The edge is implemented as a struct



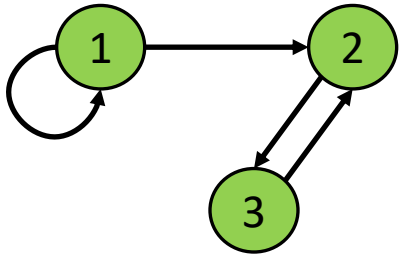


We also need to have a struct for the actual list (can also be used as a node pointer)

Representation – Successor list

Indicates for each node which neighbors it has (successors on a directed graph)

The whole graph is thus represented in one and the same amount



{(1, [1,2]), (2, [3], (3, [2]))}

Node 1 has an edge with 1 and 2

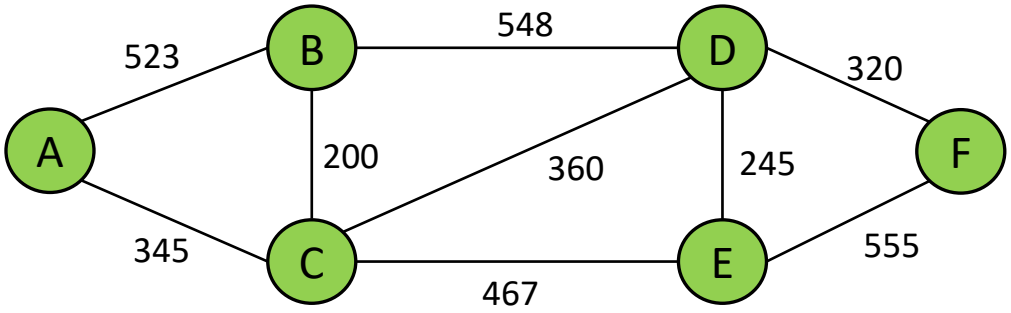
Node 2 has an edge with 3

Node 3 has an edge with 2

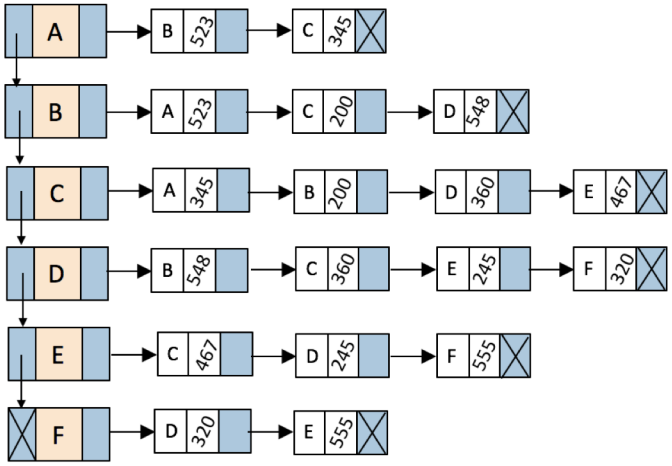
Node 2 and 3 have edges with each other

Network - weighted graphs

A network is a graph where the edges have weight



For example, to specify distance, cost etc.

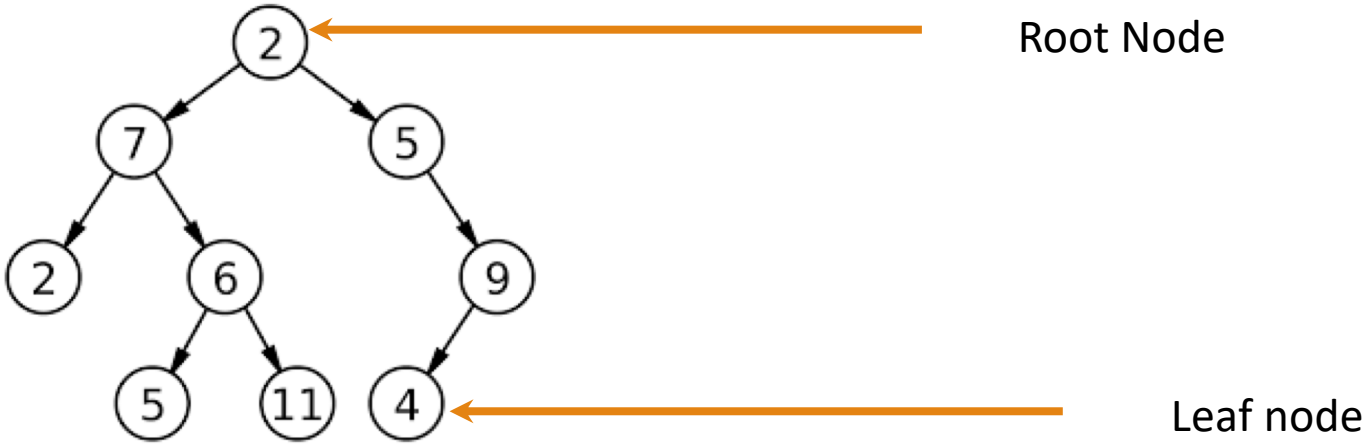


	A	B	C	D	E	F
A	0	523	345	0	0	0
B	523	0	200	548	0	0
C	345	200	0	360	467	0
D	0	548	360	0	245	320
E	0	0	467	245	0	555
F	0	0	0	320	555	0

Tree

A tree is an abstract data type. It consists of the following:

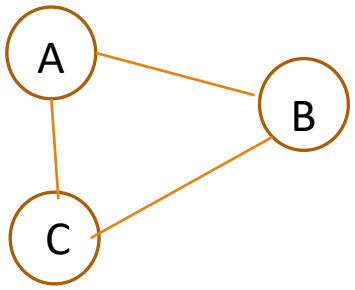
- It has an unique node called root
- Each node may have 0 or more sucessors



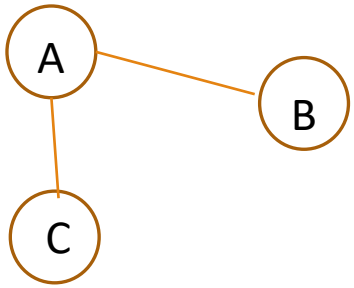
Unlike graph, a tree has no cycles

Spanning Tree

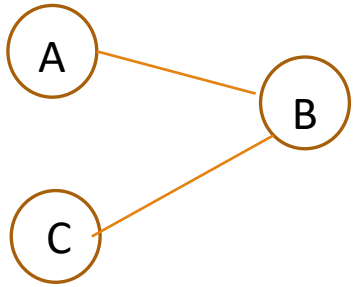
- A spanning tree is a subgraph T of Graph G such that
  - T is a tree, and
  - T covers all the vertices of G



Graph G



Spanning Tree T1



Spanning Tree T2

Minimum Spanning Tree

A minimum spanning tree (MST) is a spanning tree of any graph G in which the sum of the edge weights are minimum when compared with the sum of the edge weights of all other spanning trees of G.

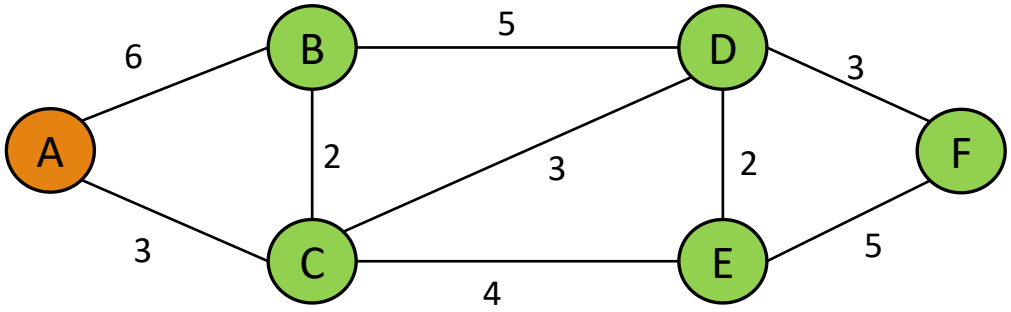
- It ensures that all nodes in a network are paired effectively
- We may not get the shortest path between two specific nodes in the graph (Shortest path tree)
- We will not go into the details- but we shall see an example

Finding Minimum Spanning Tree

Prim's Algorithm:

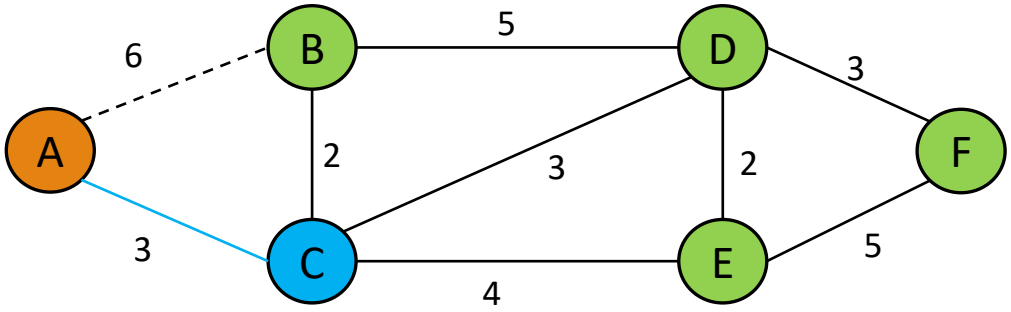
1. Start any node of the graph as the root of the MST
2. Add one edge at each step: pick an edge such that one node is already in the tree, the other node is not yet in the tree and the edge weight is minimum.
3. Repeat step 2 until all nodes are included in the tree.

Example: Finding the MST



Let us choose any node/vertex. Suppose A

Example: Finding the MST

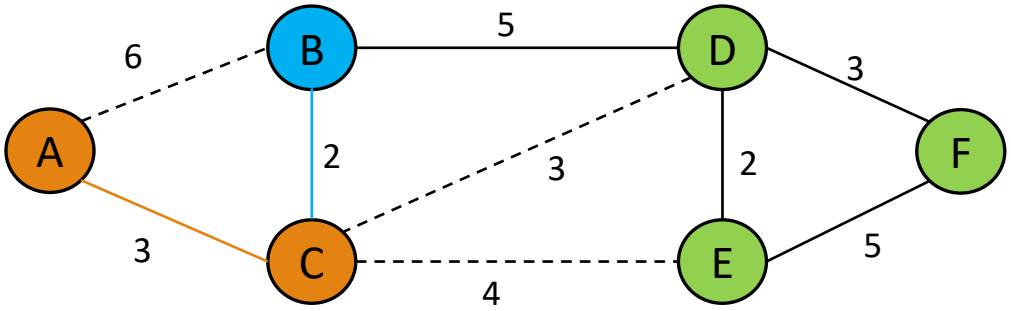


Pick an edge such that one node is already in the tree, the other node is not yet in the tree and the edge weight is minimum.

First we choose A.

Next, we choose an edge between AB and AC and include C in the MST

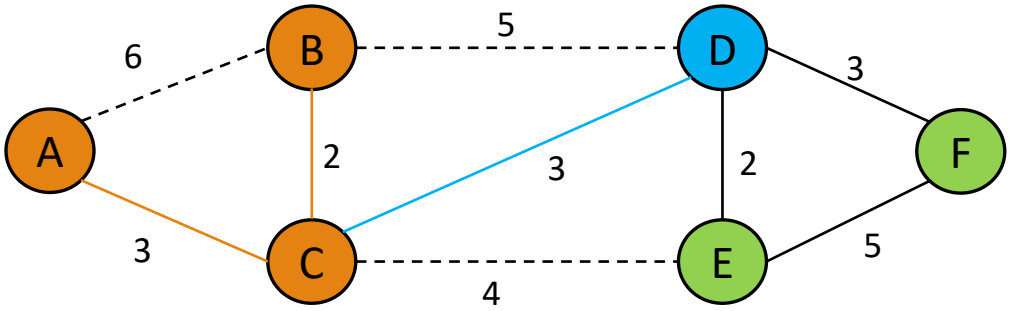
Example: Finding the MST



We now have the following options:

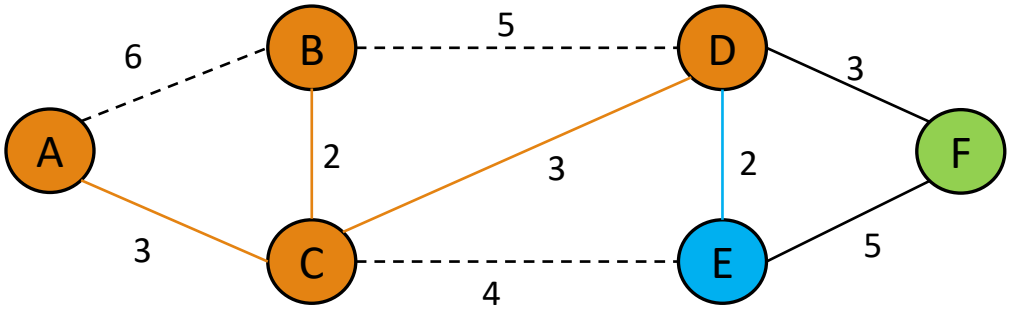
1. (A,B) with weight 6
2. (C,B) with weight 2. (we choose the minimum)
3. (C,D) with weight 3
4. (C,E) with weight 4

Example: Finding the MST



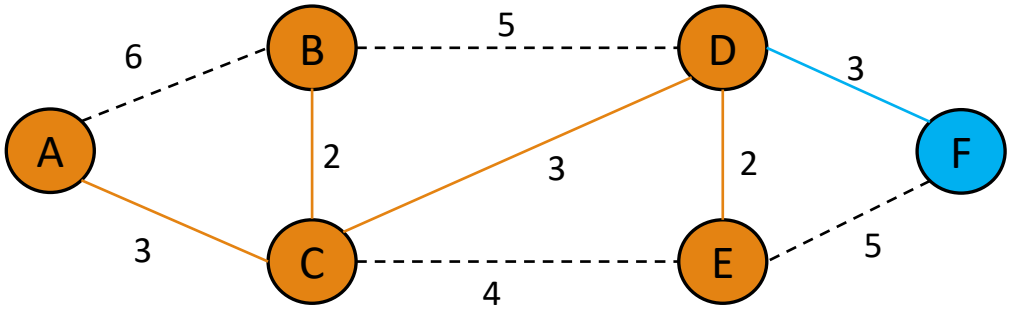
We continue choosing a minimum weighted edge such that one Node is already in the tree and the other node is not yet included in the tree.

Example: Finding the MST



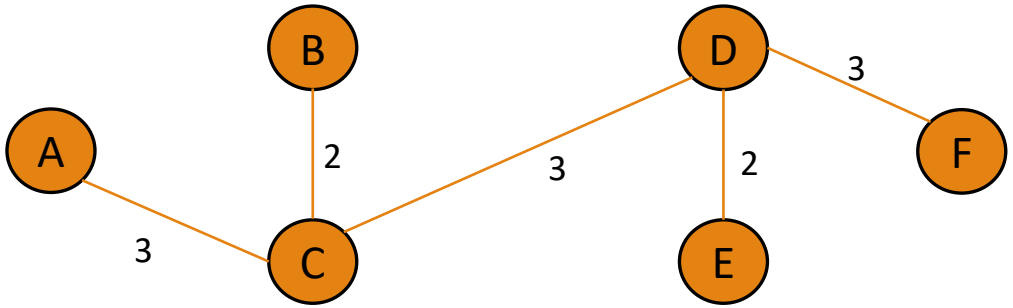
We continue choosing a minimum weighted edge such that one Node is already in the tree and the other node is not yet included in the tree.

Example: Finding the MST



We continue choosing a minimum weighted edge such that one Node is already in the tree and the other node is not yet included in the tree.

Example: Finding the MST



This approach is callled greedy method

Single Source Shortest Path

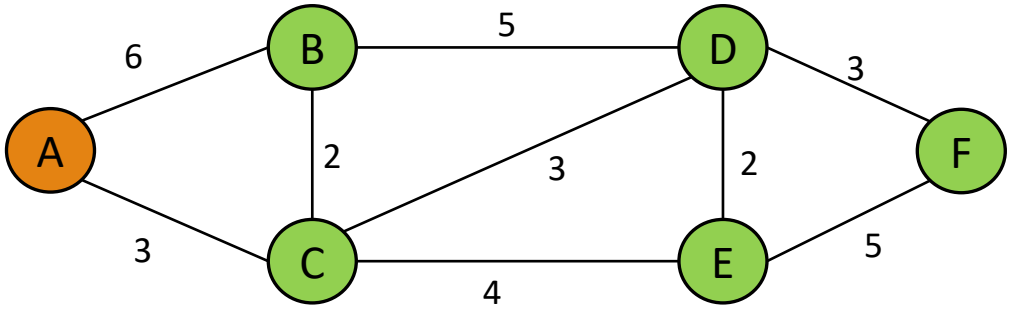
Problem: Find the shortest path from a given node to the rest of the nodes in the graph (shortest path tree)

Dijkstra Algorithm

- Using an algorithm can we find a Shortest path tree from MST- the sum of the weights between two nodes is guaranteed to be the minimum
  - Very useful for example plan road networks, travel routes or costs.
  - We will not go into the details. Only see an example

Shortest Path Tree

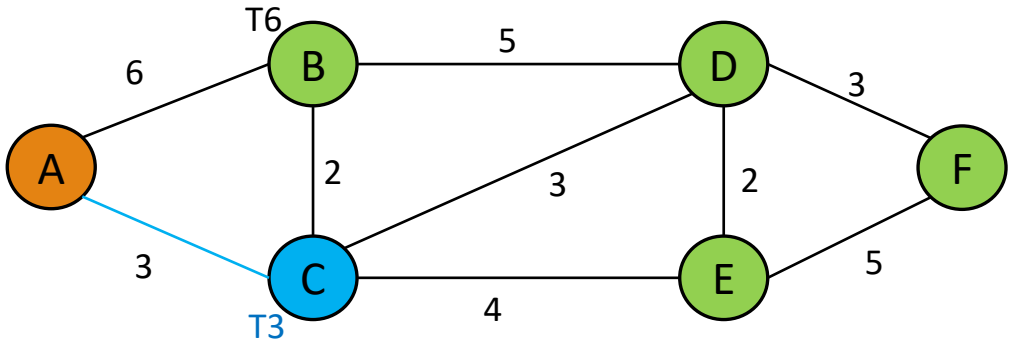
We constantly look at the total weight from the given node to the nodes chosen next.



Suppose we consider A as the starting node.

Shortest path tree

We constantly look at the total weight from the given node to the nodes chosen next.

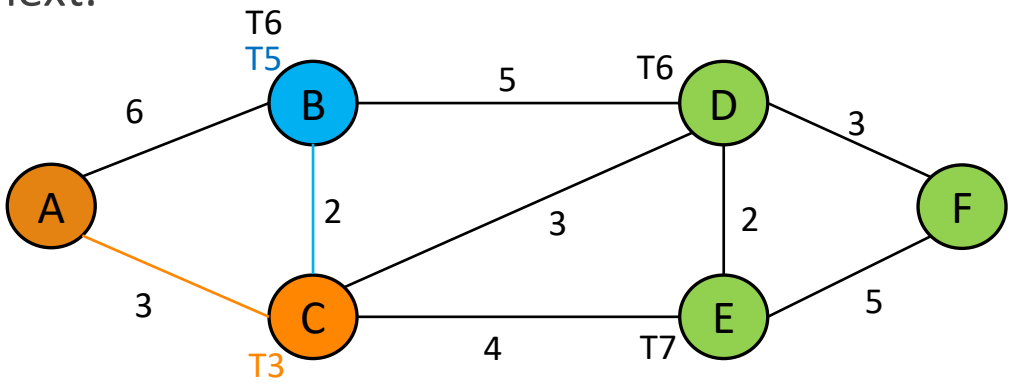


Suppose we consider A as the starting node.

Select nodes connected with A having minimum total weight- AB has total weight 6, AC has total weight 3 - select AC

Shortest path tree

We constantly look at the total weight from the given node to the nodes chosen next.



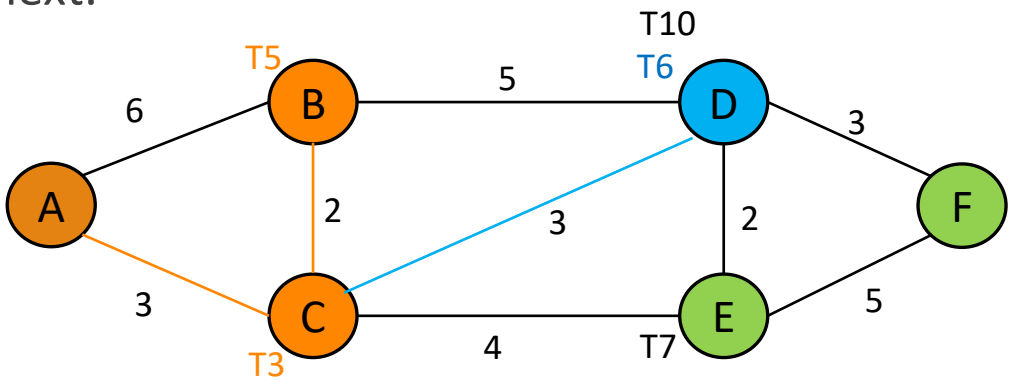
Suppose we consider A as the starting node.

Select nodes connected with A having minimum total weight- AB has total weight 6, AC has total weight 3 - select AC

Choose the lowest total weight from C - AB has T6, CB has T5, CD has T6, CE has T7 - choose CB

Shortest path tree

We constantly look at the total weight from the given node to the nodes chosen next.



Suppose we consider A as the starting node.

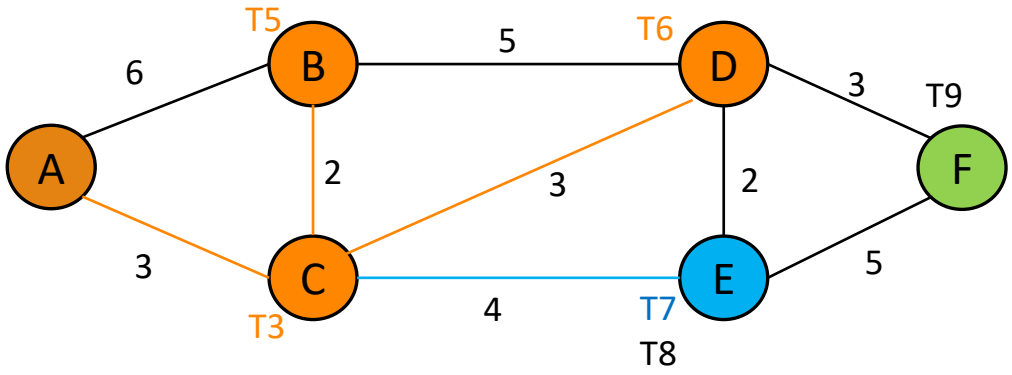
Select nodes connected with A having minimum total weight- AB has total weight 6, AC has total weight 3 - select AC

Choose the lowest total weight from C - AB has T6, CB has T5, CD has T6, CE has T7 - choose CB

Select the lowest total weight from B or C - BD has T10, CD has T6, CE has T7 - Select CD

Shortest path tree

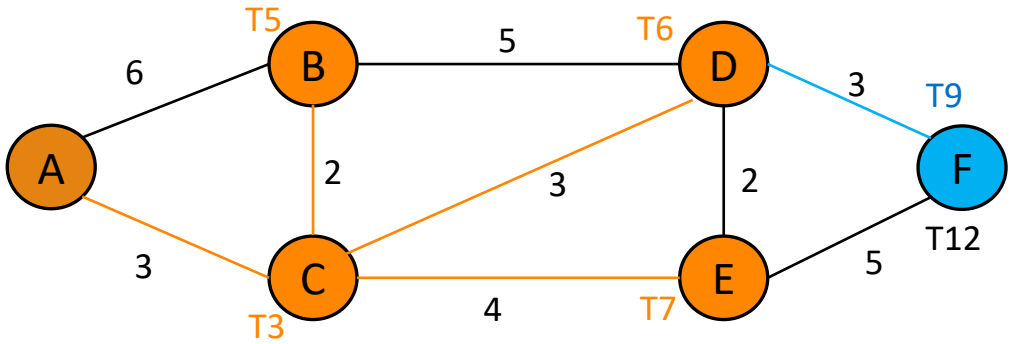
We constantly look at the total weight from the given node to the nodes chosen next.



Choose the lowest total weight from C or D - CE has T7, DE have T8, DF has T9 - choose CE

Shortest path tree

We constantly look at the total weight from the given node to the nodes chosen next.



Choose the lowest total weight from D or E - DF has T9, EF has T12 - select DF

Shortest path tree

We constantly look at the total weight from the given node to the nodes chosen next.