COMP1038.PGA

Session 17 & 18: Stack, Queue and Other data structures

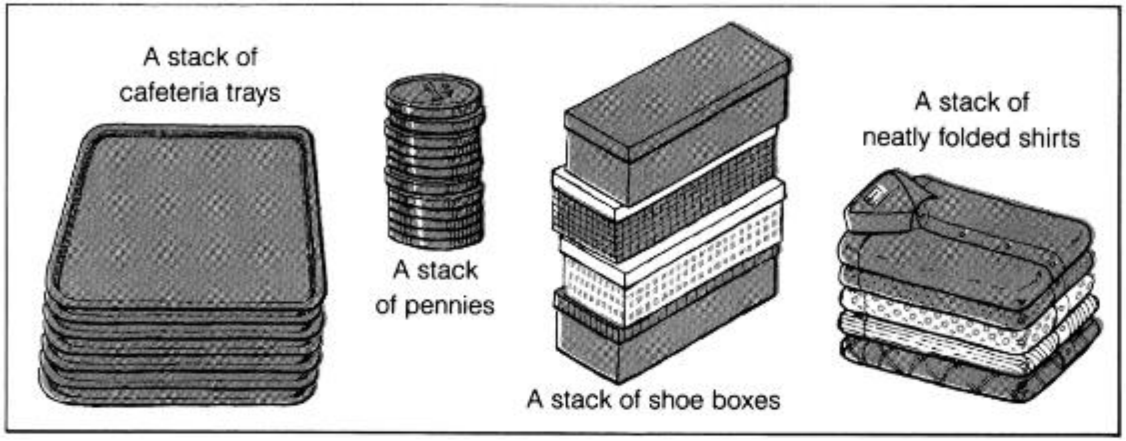
Overview

- Stack
  - Introduction
  - Creation
  - Push operation
  - Pop operation
  - Application
- Queue
  - Introduction
  - Creation
  - Enqueue
  - Dequeue
  - Application
- Tree
- Graph

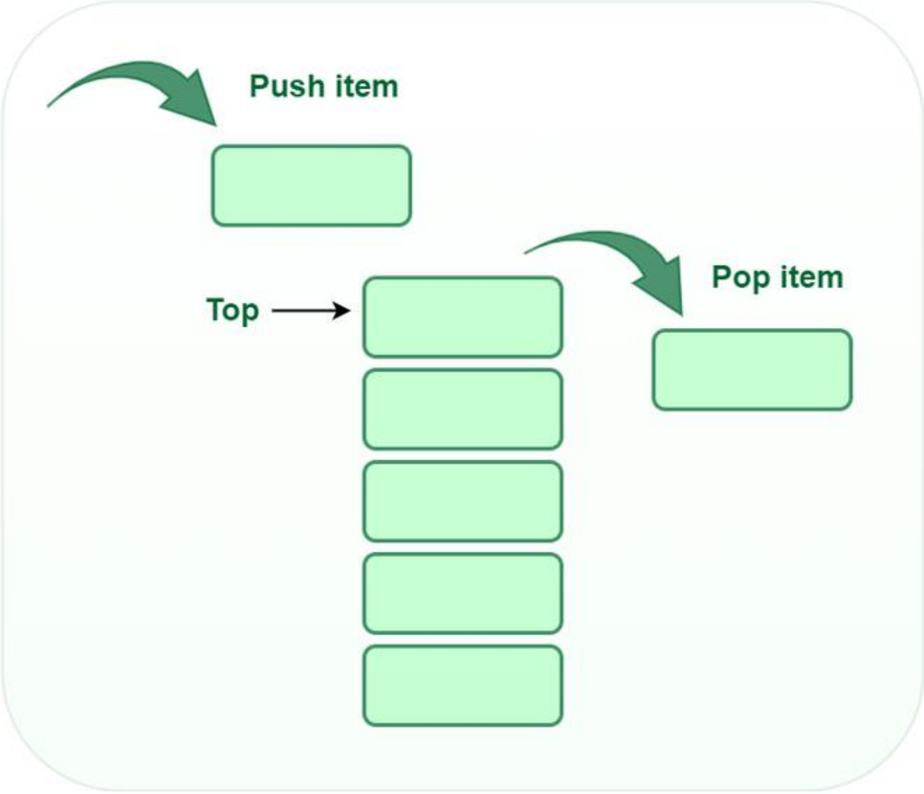
Stack

Stack: Introduction

- It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (LIFO: Last in, First Out).



Stack: Introduction cont…



Stack: basic operations

- Basic Stack Operations:
  - Initialize the Stack
  - Pop an item off the top of the stack (delete an item)
  - Push an item onto the top of the stack (insert an item)
  - Is the stack empty?
  - Is the stack full?
  - Determine the stack size.

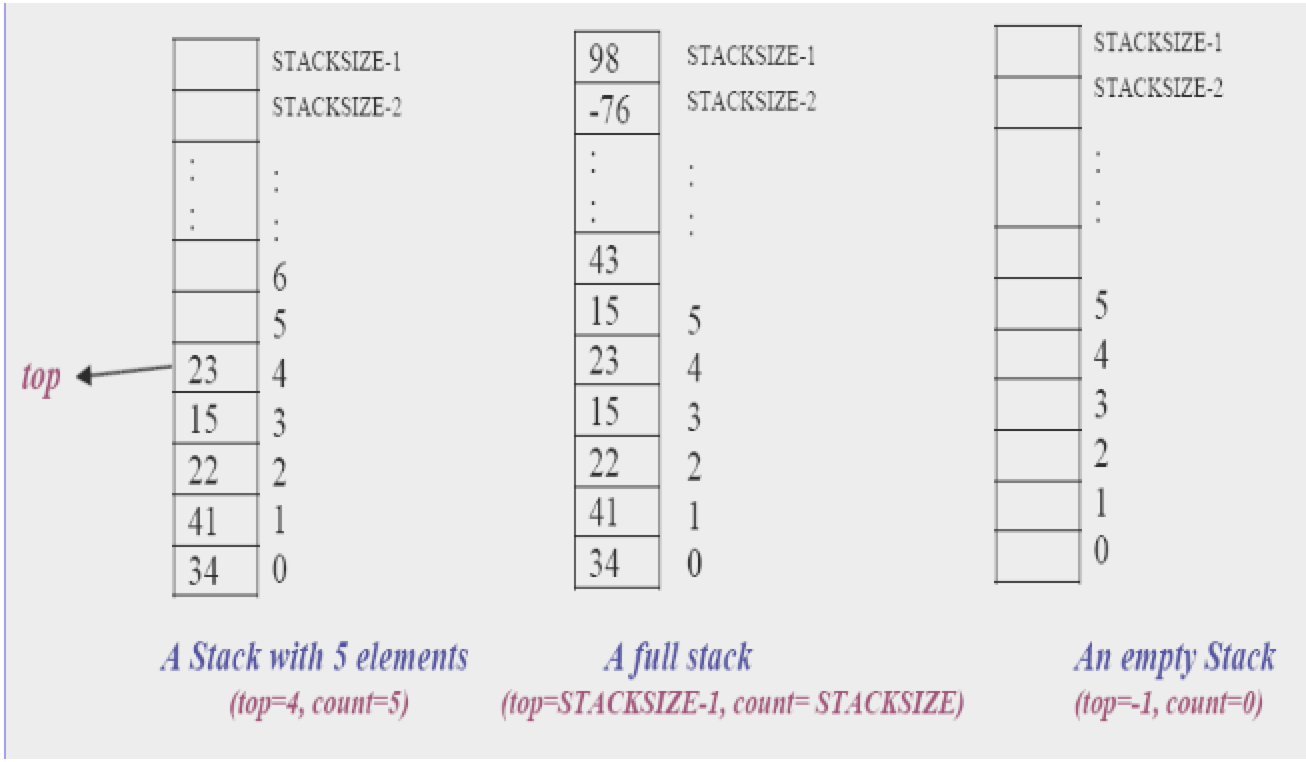
Stack: creation

- The stacks can be created by the use of **Arrays** or **Linked lists.**
  - One way to create stack is to have a data structure where a variable called top keeps the location of the elements in the stack (array)
  - An array is used to store the elements in the stack

Stack: creation using Array

```
struct Stack
{
   unsigned capacity; // keeps the number of elements in the stack.
   int top;  //indicates the location of the top of the stack
   int *array;  //pointer to the array to store the stack elements
};

struct Stack* createStack(unsigned capacity)
{
   struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
   stack->capacity = capacity;
   stack->top = -1;
   stack->array = (int*)malloc(stack->capacity * sizeof(int));
   return stack;
}
```

Stack: creation using Array cont...



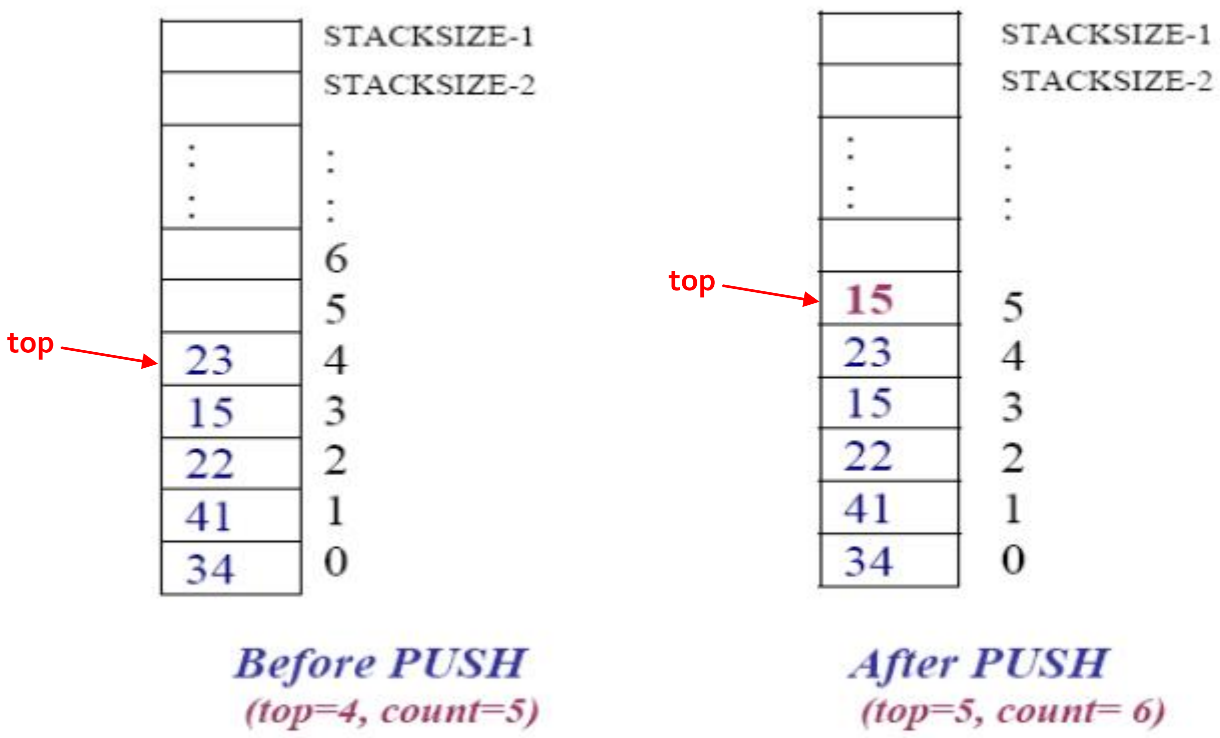
The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

Stack: isFull or isEmpty

```
// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
   return stack->top == stack->capacity - 1;
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
   return stack->top == -1;
}
```

Stack: Push operation

Push an item onto the top of the stack (insert an item)



Stack: Push operation

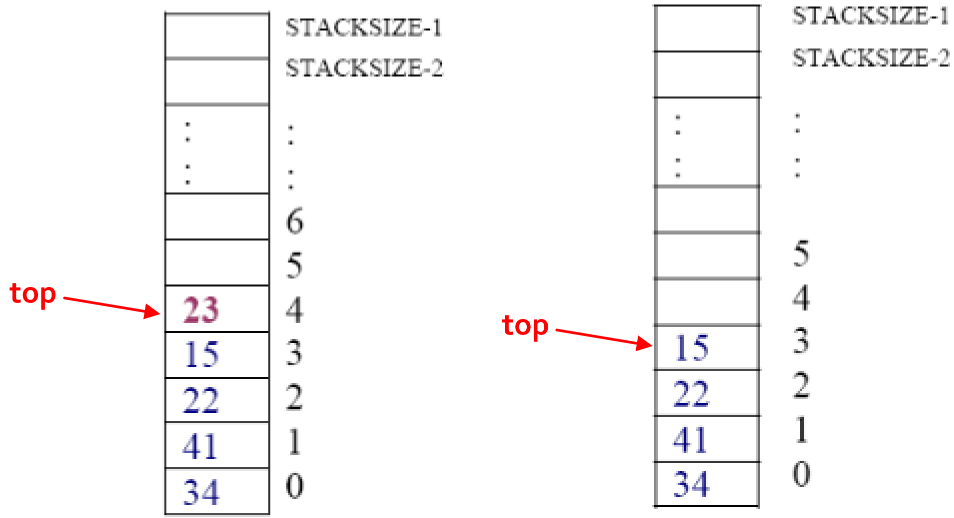
- *Function*: Adds new item to the top of the stack
- *Precondition*: stack has been initialized and is not full.
- *Post condition:* new item is at the top of the stack

Stack: push operation cont...

```
// Function to add an item to stack.  It increases top by 1
void push(struct Stack* stack, int item)
{
   if (isFull(stack))
      return;
   stack->array[++stack->top] = item;
   printf("%d pushed to stack\n", item);
}
```

Stack: pop operation

- Pop an item off the top of the stack (delete an item)



Before POP
(top=4, count=5)

After POP
(top=3 count=4)

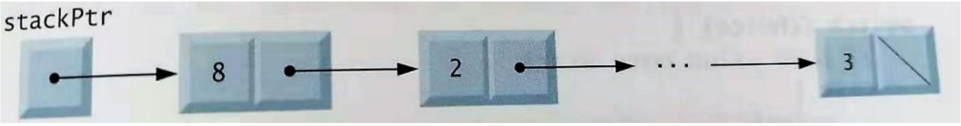
Stack: pop operation cont..

- *Function:* Removes top item from stack and returns with top item.
- *Preconditions:* Stack has been initialized and is not empty.
- *Post conditions:* Top element has been removed from stack and the function returns with the top element.

Stack: pop operation cont..

```
// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack)
{
   if (isEmpty(stack))
      return -1;
   return stack->array[stack->top--];
}
```

Stack: creation using Linked list



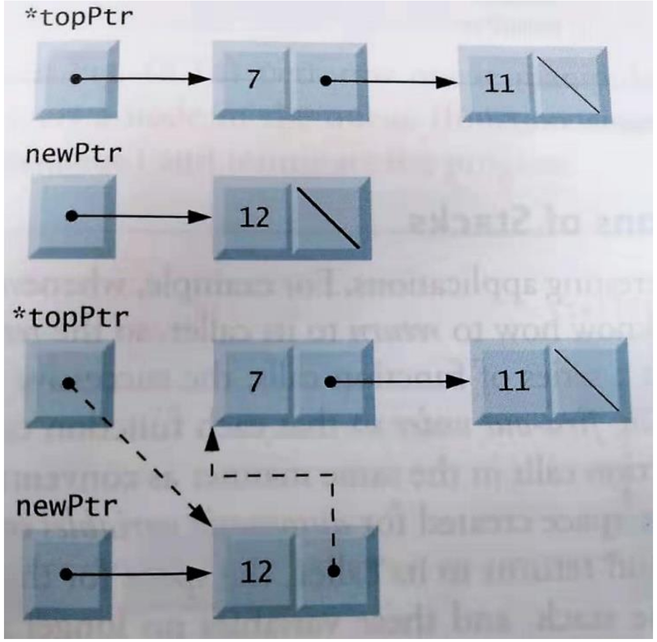

```
struct StackNode {
   int data;
   struct StackNode* next;
};

struct StackNode* newNode(int data)
{
   struct StackNode* stackNode = (struct
StackNode*)malloc(sizeof(struct StackNode));
   stackNode->data = data;
   stackNode->next = NULL;
   return stackNode;
}
```

Stack: isEmpty

```
int isEmpty(struct StackNode* topPtr)
{
    return ! topPtr;
}
```

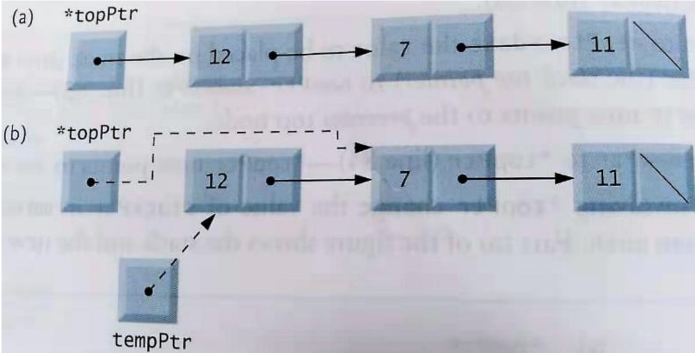
Stack: push operation

```
void push(struct  StackNode** topPtr, int data)
{
   struct  StackNode* newPtr = newNode(data);
   newPtr ->next = * topPtr;
   * topPtr = newPtr;
   printf("%d pushed to stack\n", data);
}
```



Stack: pop operation

```
int pop(struct StackNode** topPtr)
{
   if (isEmpty(* topPtr))
      return -1;
   struct  StackNode* tempPtr = * topPtr;
   * topPtr = (* topPtr)->next;
   int popped = tempPtr ->data;
   free(tempPtr);

   return popped;
}
```



The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

Stack: applications

- Expression evaluation
- Expression conversion: prefix to infix, postfix to infix, infix to prefix, and infix to postfix
- Argument passing in C
- Parsing

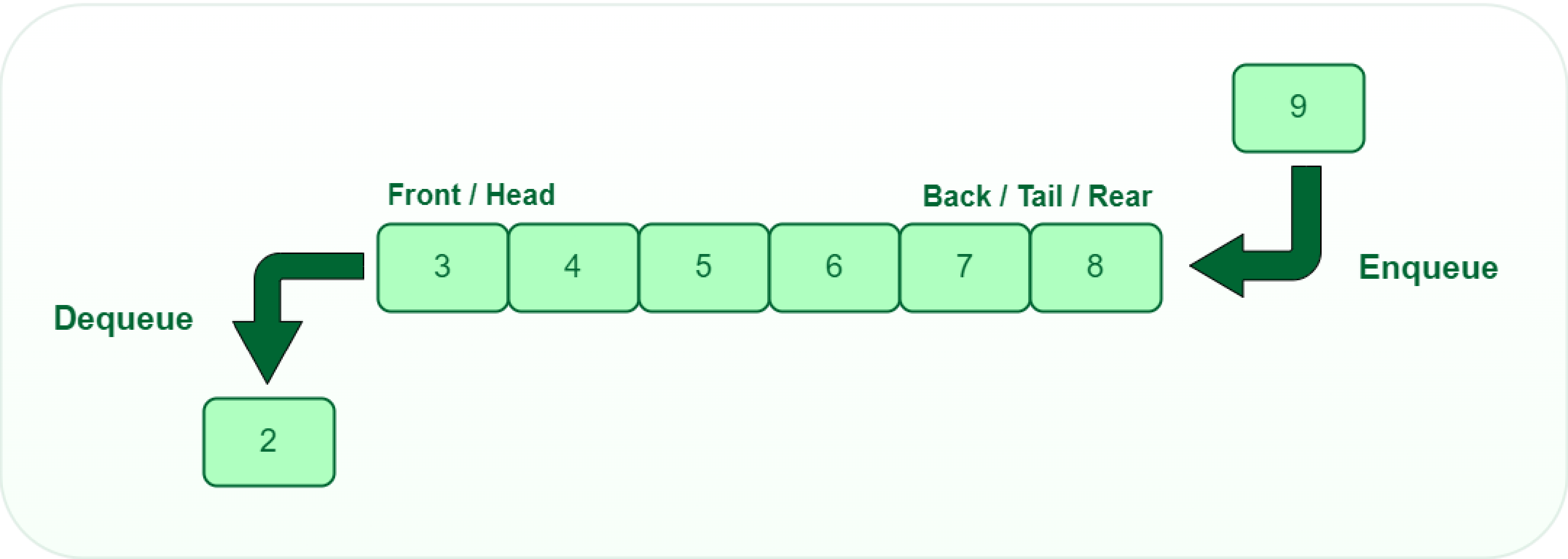
Queue

Queue: introduction

- Unlike stacks, a queue is open at both its ends.
- One end (rear/tail end) is always used to insert data (enqueue) and the other (front/head end) is used to remove data (dequeue).
- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Queue: introduction cont…

Queue: introduction cont…



Queue: basic operations

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.
- **isFull()** – Checks if the queue is full.
- **isEmpty()** – Checks if the queue is empty.

Queue: creation with array

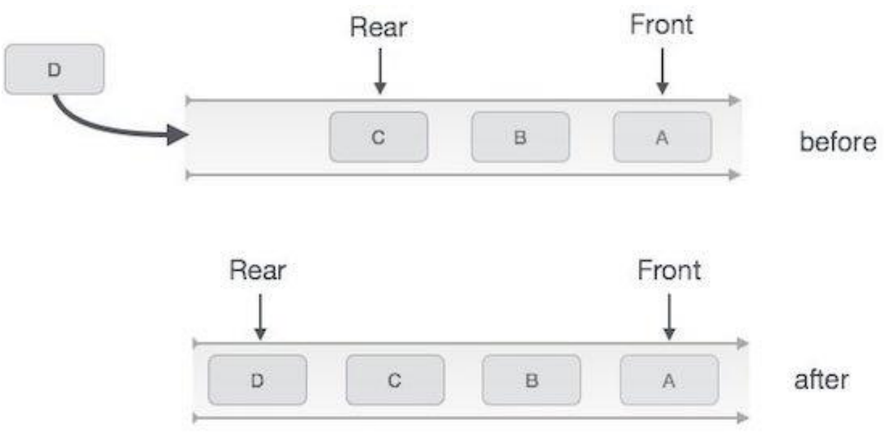
```
// A structure to represent a queue
struct Queue
{
   int front, rear, size;
   unsigned capacity;
   int* array;
};

// function to create a queue of given capacity.
// It initializes size of queue as 0
struct Queue* createQueue(unsigned capacity)
{
   struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue));
   queue->capacity = capacity;
   queue->front = queue->size = 0;
   queue->rear = capacity - 1;  // This is important, see the enqueue
   queue->array = (int*) malloc(queue->capacity * sizeof(int));
   return queue;
}
```

Queue: isFull() and isEmpty()

```
// Queue is full when size becomes equal to the capacity
int isFull(struct Queue* queue)
{  return (queue->size == queue->capacity);
}

// Queue is empty when size is 0
int isEmpty(struct Queue* queue)
{  return (queue->size == 0);
}
```

Queue: enqueue operation

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success.



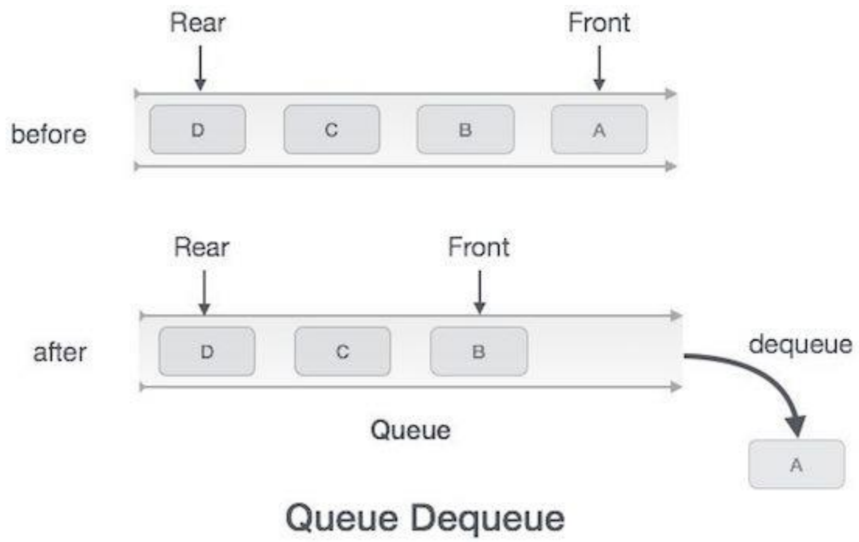
Queue Enqueue

Queue: enqueue operation cont…

```
// Function to add an item to the queue.
// It changes rear and size

void enqueue(struct Queue* queue, int item)
{
   if (isFull(queue))
      return;
   queue->rear = (queue->rear + 1)%queue->capacity;
   queue->array[queue->rear] = item;
   queue->size = queue->size + 1;
   printf("%d enqueued to queue\n", item);
}
```

Queue: dequeue operation

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.

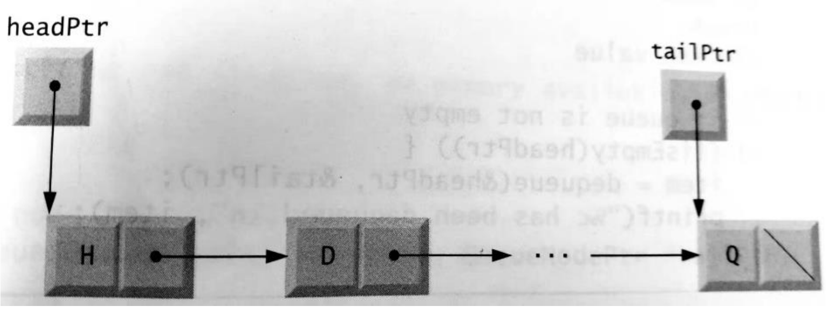


Queue: dequeue operation cont…

```
// Function to remove an item from queue.
// It changes front and size
int dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1)%queue->capacity;
    queue->size = queue->size - 1;
    return item;
}
```

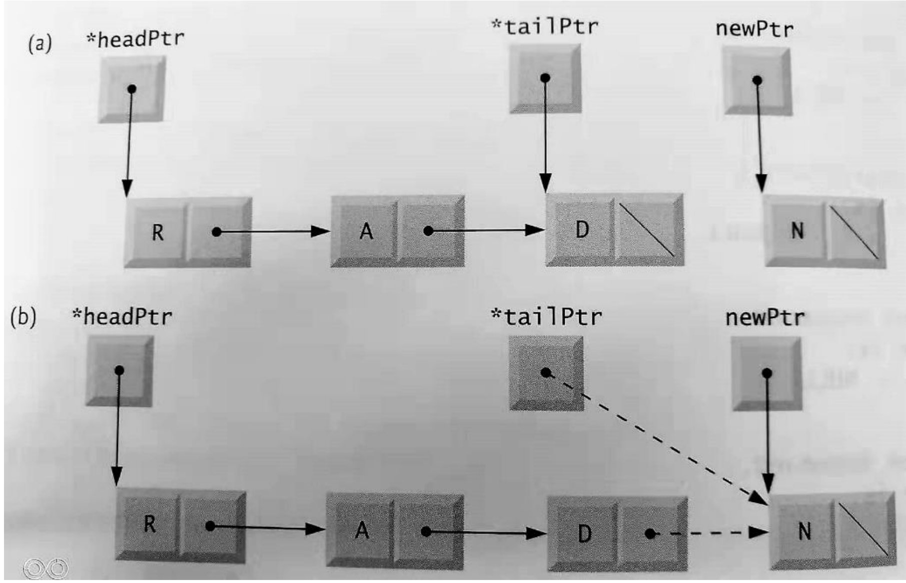
Queue: creation with linked list

```
// A linked list (LL) node to store a queue entry
struct QNode {
   int key;
   struct QNode* next;
};

// The queue, front stores the front node of LL and rear stores the
// last node of LL
struct Queue {
   struct QNode *headPtr, * tailPtr;
};

// A utility function to create a new linked list node.
struct QNode* newNode(int k)
{
   struct QNode* tempPtr = (struct QNode*)malloc(sizeof(struct QNode));
   tempPtr->key = k;
   tempPtr->next = NULL;
   return tempPtr;
}

// A utility function to create an empty queue
struct Queue* createQueue()
{
   struct Queue* newPtr = (struct Queue*)malloc(sizeof(struct Queue));
   newPtr -> headPtr = newPtr ->tailPtr = NULL;
   return newPtr;
}
```



Queue: enqueuer operation

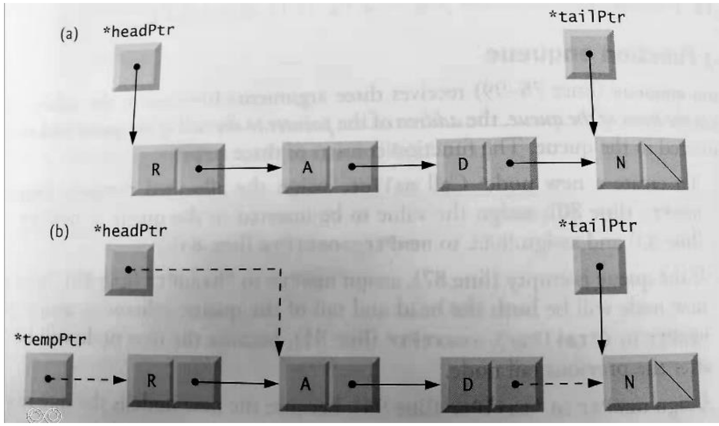
```
// The function to add a key k to q
void enqueue(struct Queue* q, int k)
{
  // Create a new LL node
  struct QNode* newPtr = newNode(k);

   // If queue is empty, then new node is front and rear both
  if (q->headPtr == NULL) {
    q-> headPtr = q->tailPtr = newPtr;
    return;
  }

  // Add the new node at the end of queue and change rear
  q-> tailPtr ->next = newPtr;
  q-> tailPtr = newPtr;
}
```



Queue: dequeue operation

```
// Function to remove a key from given queue q
struct QNode* dequeue(struct Queue* q)
{
  // If queue is empty, return NULL.
  if (q->headPtr == NULL)
    return NULL;

  // Store previous front and move front one node ahead
  struct QNode* tempPtr = q->headPtr;

  q-> headPtr = q-> headPtr ->next;

   // If front becomes NULL, then change rear also as NULL
  if (q-> headPtr == NULL)
    q->tailPtr = NULL;
  return tempPtr;
}
```

```
//Now, free the memory inside the calling function

struct QNode* dequeuenode;
dequeuenode = dequeue(q);
free(dequeuenode);
```



Queue: applications

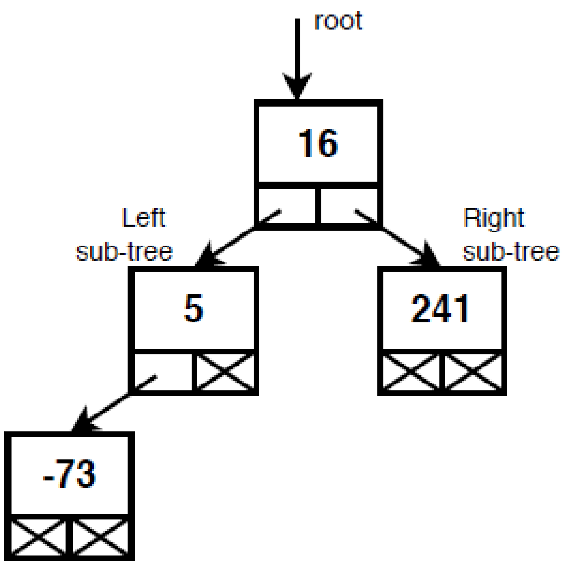
- Queue is useful in CPU scheduling, Disk Scheduling. When multiple processes require CPU at the same time, various CPU scheduling algorithms are used which are implemented using Queue data structure.
- When data is transferred asynchronously between two processes. Queue is used for synchronization. Examples : IO Buffers, pipes, file IO, etc.
- Print Spooling. In print spooling, documents are loaded into a buffer and then the printer pulls them off the buffer at its own rate. Spooling also lets you place a number of print jobs on a queue instead of waiting for each one to finish before specifying the next one.
- Breadth First search in a Graph .It is an algorithm for traversing or searching graph data structures. It starts at some arbitrary node of a graph and explores the neighbor nodes first, before moving to the next level neighbors.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
- In real life, Call Center phone systems use Queues, to hold people calling them in an order, until a service representative is free.

Thank you!

Tree

Tree: introduction

- Trees are hierarchical data structures containing nodes which store a value and references to 2 or more subtrees.
- The "start" of the tree is the root node.
- Nodes with no subtrees are called leaf nodes.
- A binary tree is a tree where each node has exactly two possible children.



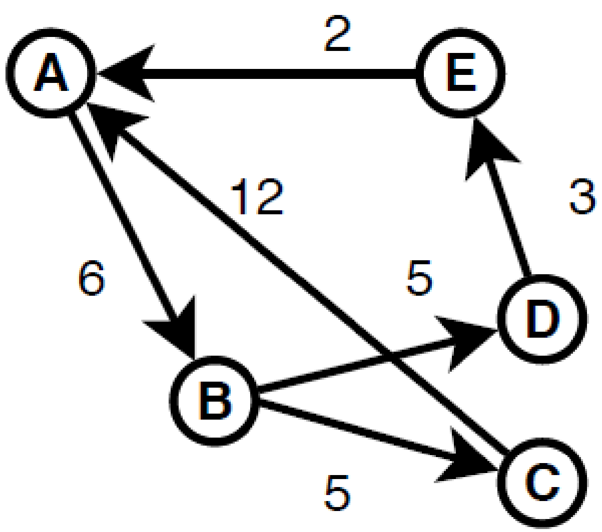
Tree: binary search tree

- BSTs are binary trees in which the values are stored in the tree in some specified order.
- Eg, for an integer BST,
  - every value in the left sub-tree < value in the node
  - every value in the right sub-tree >= the value in the node.
- Searching for values in a BST can be extremely quick because each comparison discards half the remaining values (on average).
- Inserting/removing nodes is more complex as it may require moving existing nodes.

Graph

Graph: introduction

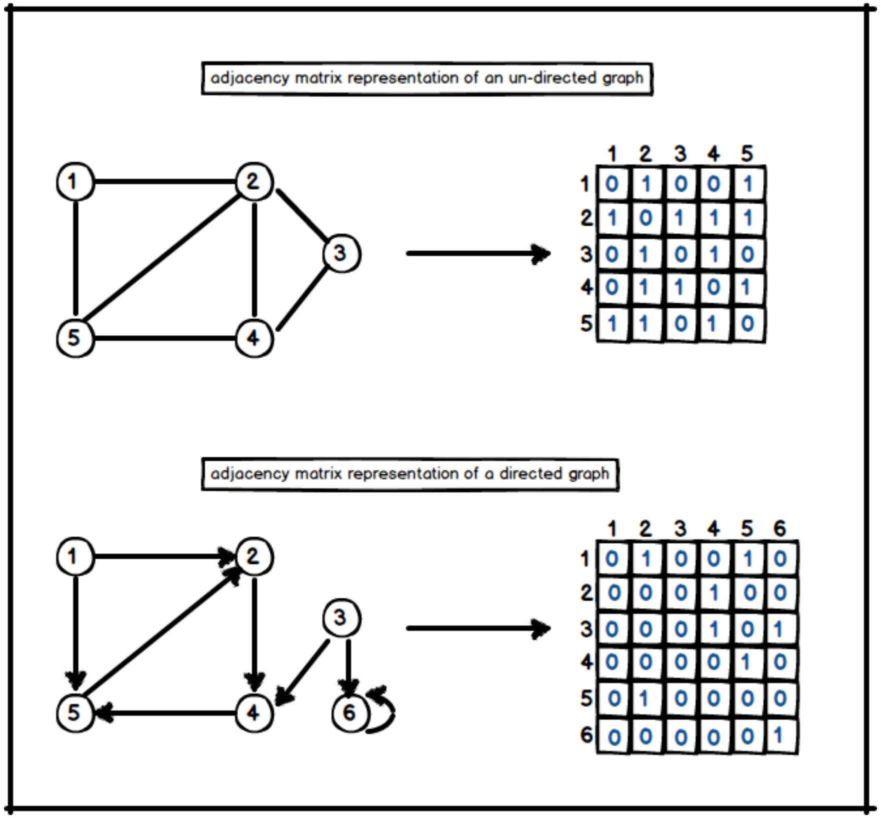
- A graph represents a set of vertices (or nodes) and a set of edges which are connections between vertices.
- Edges can be directed or undirected.
- Edges can be weighted or unweighted.
- Graphs can be connected (for any 2 vertices, there is a path between them) or unconnected.
- ...and many other possible properties
  - Graph theory is a major branch of discrete maths.



Graph: adjacency matrix

- Vertices are assigned to integer IDs.
- Store graph as a 2D array of integers.
- A_{ij} contains a value if there is an edge from i to j.
  - Store edge present? Number of edges? Weight of edge?
- Very fast look-up for edge between two vertices.
- Low memory usage but wastes lots of space for sparse graphs.
- Requires whole array to be changed when adding/removing vertices.

Graph: adjacency matrix cont…



Graph: applications

- Modeling relationships between users in social media networks (vertex=user, edge=friends).
- Navigation through cities vertex=road junction, edge=road between junction, weight=distance).
- Modelling state machines (vertex=state, edge=valid transition).

Thank you!