Stacks and Queues

Akash Hegde

Seventh Sense Talent Solutions

Vivekananda Institute of Technology

25 March 2021

Introduction to Stack

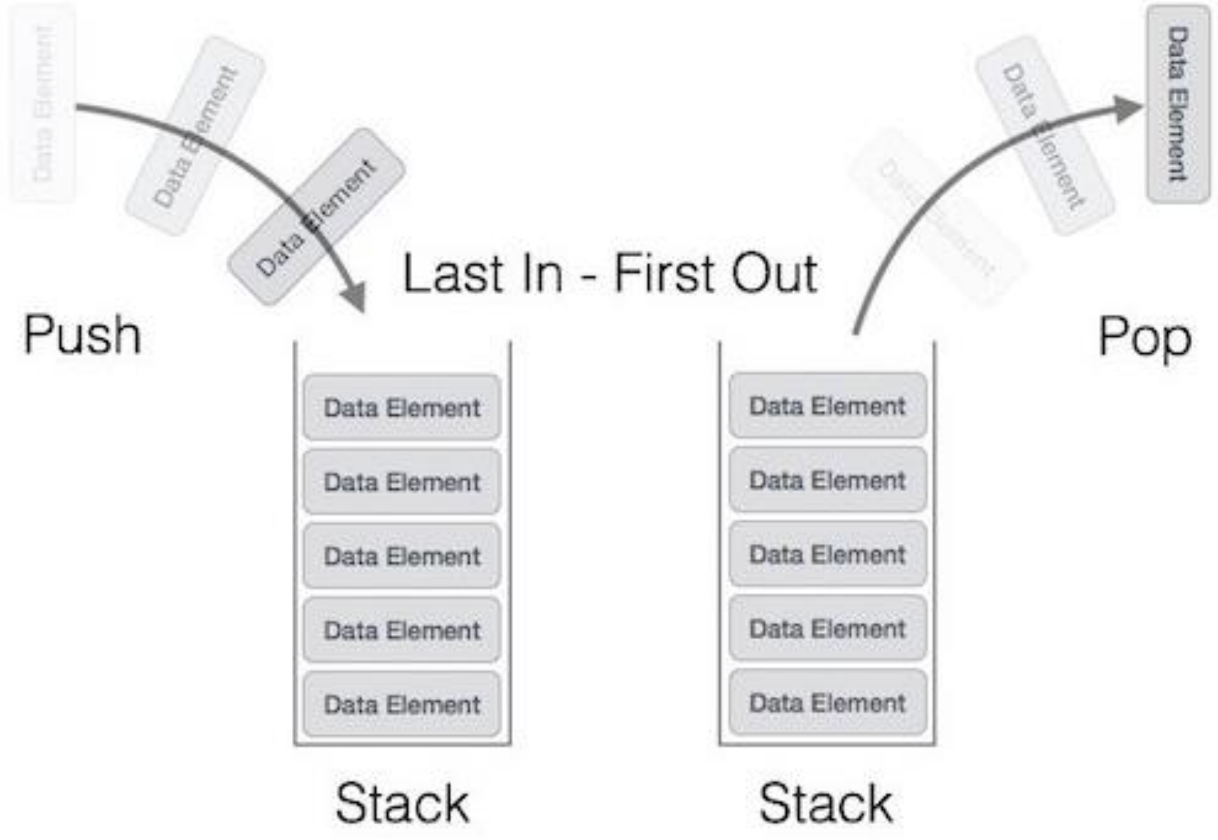
- A **stack** is an Abstract Data Type (ADT), commonly used in most programming languages.
- It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates.
- A real-world stack allows operations at one end only.
- For example, we can place or remove a card or plate from the top of the stack only.
- Likewise, Stack ADT allows all data operations at one end only.
- At any given time, we can only access the top element of a stack.

Introduction to Stack

- This feature makes it LIFO data structure.
- LIFO stands for Last-in-first-out.
- Here, the element which is placed (inserted or added) last, is accessed first.
- In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

- The following diagram depicts a stack and its operations –



Stack Operations

- Stack operations may involve initializing the stack, using it and then de-initializing it.
- Apart from these basic operations, a stack is used for the following two primary operations −
- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

Stack Operations

- To use a stack efficiently, we need to check the status of stack as well.
- For the same purpose, the following functionality is added to stacks –
- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

Stack Operations

- At all times, we maintain a pointer to the last pushed data on the stack.
- As this pointer always represents the top of the stack, hence named **top**.
- The **top** pointer provides top value of the stack without actually removing it.

Stack Operations – peek()

- Algorithm of peek() function –
  `begin procedure peek`
- **`    return stack[top]`**
- **`end procedure`**
- Implementation of peek() function –
  `int peek() {`
- `    return stack[top];`
- `}`

Stack Operations – isfull()

- Algorithm of isfull() function –
  begin procedure isfull
- **if top equals to MAXSIZE**
- **return true**
- **else**
- **return false**
- **endif**
- **end procedure**
- Implementation of isfull() function –
  bool isfull() {
- if(top == MAXSIZE)
- return true;
- else
- return false;

Stack Operations – isempty()

- Algorithm of isempty() function –
  begin procedure isempty
- **if top less than 1**
- **return true**
- **else**
- **return false**
- **endif**
- **end procedure**
- Implementation of isempty() function -
  bool isempty() {
- if(top == -1)
- return true;
- else
- return false;

Stack Operations – push()

- The process of putting a new data element onto stack is known as a Push Operation.
- Push operation involves a series of steps –
  - **Step 1** – Checks if the stack is full.
  - **Step 2** – If the stack is full, produces an error and exit.
  - **Step 3** – If the stack is not full, increments **top** to point next empty space.
  - **Step 4** – Adds data element to the stack location, where top is pointing.
  - **Step 5** – Returns success.

Stack Operations – push()

- Algorithm for PUSH Operation
  begin procedure push: stack, data
- **if stack is full**
- **return null**
- **endif**
- **top ← top + 1**
- **stack[top] ← data**
- **end procedure**
- Implementation of push() –

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
```

Stack Operations – pop()

- Accessing the content while removing it from the stack, is known as a Pop Operation.
- In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value.
- But in linked-list implementation, pop() actually removes data element and deallocates memory space.
- A Pop operation may involve the following steps −
  - **Step 1** − Checks if the stack is empty.
  - **Step 2** − If the stack is empty, produces an error and exit.
  - **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
  - **Step 4** − Decreases the value of top by 1.
  - **Step 5** − Returns success.

Stack Operations – pop()

- Algorithm for Pop Operation
```
begin procedure pop: stack
   if stack is empty
      return null
   endif
   data ← stack[top]
   top ← top - 1
   return data
end procedure
```
- Implementation of pop() –
```
int pop(int data) {
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
```

Stack Implementation

- Program to implement stack (part 1) –

```
#include <stdio.h>

int MAXSIZE = 8;
int stack[8];
int top = -1;

int isempty() {

   if(top == -1)
      return 1;
   else
      return 0;
}

int isfull() {

   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

int peek() {
   return stack[top];
}
```

Stack Implementation

- Program to implement stack (part 2) –

```
int pop() {
   int data;

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}

int push(int data) {

   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Stack Implementation

- Program to implement stack (part 3) –

```c
int main() {
   // push items on to the stack
   push(3);
   push(5);
   push(9);
   push(1);
   push(12);
   push(15);

   printf("Element at top of the stack: %d\n" ,peek());
   printf("Elements: \n");

   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d\n",data);
   }

   printf("Stack full: %s\n" , isfull()?"true":"false");
   printf("Stack empty: %s\n" , isempty()?"true":"false");

   return 0;
}
```

Introduction to Queue

- Queue is an abstract data structure, somewhat similar to Stacks.
- Unlike stacks, a queue is open at both its ends.
- One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).
- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.
- A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

- The following diagram gives queue representation as data structure –

Queue Operations

- Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory.
- Here we shall try to understand the basic operations associated with queues –
- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Queue Operations

- Few more functions are required to make the above-mentioned queue operation efficient. These are –
- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.
- In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueuing (or storing) data in the queue we take help of **rear** pointer.

Queue Operations – peek()

- This function helps to see the data at the **front** of the queue.
- The algorithm of peek() function is as follows –
  begin procedure peek
- **return queue[front]**
- **end procedure**
- Implementation of peek() function -
  int peek() {
-     return queue[front];
- }

Queue Operations – isfull()

- Algorithm:
  begin procedure isfull
-     if rear equals to MAXSIZE
-         return true
-     else
-         return false
-     endif
- end procedure
- Implementation of isfull() function –
  bool isfull() {
-     if(rear == MAXSIZE - 1)
-         return true;
-     else
-         return false;

Queue Operations – isempty()

- Algorithm:
  begin procedure isempty
-     if front is < MIN OR front is > rear
-         return true
-     else
-         return false
-     endif
- end procedure
- Implementation:
  bool isempty() {
-     if(front < 0 || front > rear)
-         return true;
-     else
-         return false;

Queue Operations – enqueue()

- Queues maintain two data pointers, **front** and **rear**.
- Therefore, its operations are comparatively difficult to implement than that of stacks.
- The following steps should be taken to enqueue (insert) data into a queue –
  - **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 Operations – enqueue()

- Algorithm for enqueue operation
  procedure enqueue(data)
- **if queue is full**
- **return overflow**
- **endif**
- **rear ← rear + 1**
- **queue[rear] ← data**
- **return true**
- **end procedure**
- Implementation of enqueue() -
  int enqueue(int data){
- if(isfull())
- return 0;
- rear = rear + 1;

Queue Operations – dequeue()

- Dequeue Operation
- Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access.
- The following steps are taken to perform **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 Operations - dequeue()

- Algorithm for dequeue operation
  procedure dequeue
- **if queue is empty**
- **return underflow**
- **end if**
- **data = queue[front]**
- **front ← front + 1**
- **return true**
- **end procedure**
- Implementation of dequeue() -
  int dequeue() {
- if(isempty())
- return 0;
- int data = queue[front];

Queue Implementation

- Program to implement queue (part 1) –

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 6

int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;

int peek() {
   return intArray[front];
}

bool isEmpty() {
   return itemCount == 0;
}

bool isFull() {
   return itemCount == MAX;
}
```

Queue Implementation

- Program to implement queue (part 2) –

```
int size() {
   return itemCount;
}

void insert(int data) {

   if(!isFull()) {

      if(rear == MAX-1) {
         rear = -1;
      }

      intArray[++rear] = data;
      itemCount++;
   }
}

int removeData() {
   int data = intArray[front++];

   if(front == MAX) {
      front = 0;
   }

   itemCount--;
   return data;
}
```

Queue Implementation

- Program to implement queue (part 3) –

```
int main() {
   /* insert 5 items */
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);

   // front : 0
   // rear  : 4
   // ------------------
   // index : 0 1 2 3 4
   // ------------------
   // queue : 3 5 9 1 12
   insert(15);

   // front : 0
   // rear  : 5
   // ---------------------
   // index : 0 1 2 3 4  5
   // ---------------------
   // queue : 3 5 9 1 12 15

   if(isFull()) {
      printf("Queue is full!\n");
   }
```

Queue Implementation

- Program to implement queue (part 4) –

```
// remove one item
int num = removeData();

printf("Element removed: %d\n",num);
// front : 1
// rear  : 5
// --------------------
// index : 1 2 3 4  5
// --------------------
// queue : 5 9 1 12 15

// insert more items
insert(16);

// front : 1
// rear  : -1
// ----------------------
// index : 0  1 2 3 4  5
// ----------------------
// queue : 16 5 9 1 12 15

// As queue is full, elements will not be inserted.
insert(17);
insert(18);
```

Queue Implementation

- Program to implement queue (part 5) –

```
    // ----------------------
    // index : 0  1 2 3 4  5
    // ----------------------
    // queue : 16 5 9 1 12 15
    printf("Element at front: %d\n",peek());

    printf("----------------------\n");
    printf("index : 5 4 3 2  1  0\n");
    printf("----------------------\n");
    printf("Queue:  ");

    while(!isEmpty()) {
       int n = removeData();
       printf("%d ",n);
    }
}
```

Thank you!