DATA STRUCTURE
STACK

Table of Content

Implementation

Implement stack using static array

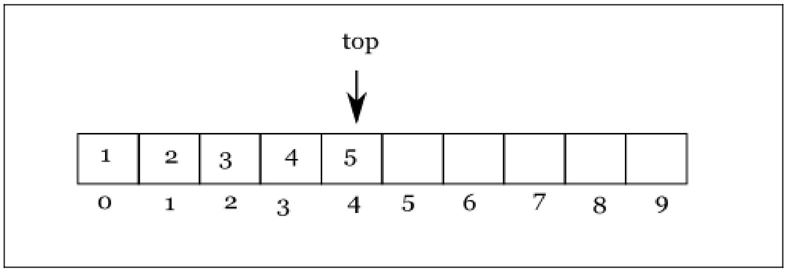
Declaration

```
#define MAX_SIZE 10

class stack

{
private:
    int arr[MAX_SIZE];
    int top = -1;

public:
    void push(int val);
    int pop();
    int peek();
    int is_empty();
};
```



Implementation

Push operation

```
void stack::push(int val)
{
    if (top == MAX_SIZE-1)
    {
        std::cout << "stack is full, can not push\n";
    }
    else
    {
        top++;
        arr[top] = val;
    }
}
```

Pop operation

```
int stack::pop()
{
    if (is_empty())
    {
        std::cout << "stack is empty, can not pop\n";
    }
    else
    {
        int val = arr[top];
        top--;
        return val;
    }
}
```

Peek operation

```
int stack::peek()
{
    if (is_empty())
    {
        std::cout << "stack is empty, can not peek\n";
        return -1;
    }
    else
    {
        return arr[top];
    }
}
```

Function is_empty()

```
int stack::is_empty()
{
    return (top == -1);
}
```

Performance

Operation	Complexity
push	O(1)
pop	O(1)
peek	O(1)

Pros
- Simple
- Operations takes constant time

Cons
- Limited size

Implement stack using dynamic array

Declaration

```
class stack
{
private:
    int *arr;
    int capacity = 5;
    int top = -1;

public:
    stack();
    ~stack();

    void push(int val);
    int pop();
    int peek();
    int is_empty();
};
```

Implementation

Constructor & destructor

```
stack::stack()
{
    arr = (int*)calloc(capacity, sizeof(int));
}

stack::~stack()
{
    free(arr);
}
```

Push operation

```
void stack::push(int val)
{
    if (top == capacity-1)
    {
        std::cout << "stack is full, expand array\n";

        capacity *= 2;
        int *tmp = (int*)realloc(arr, capacity);
        arr = tmp;
    }

    top++;
    arr[top] = val;
}
```

Performance

Operation	Complexity
push	O(1)
pop	O(1)
peek	O(1)

Pros

- Operations takes constant time
- Unlimited size

Cons

- Expanding stack is expensive.

Implement stack using linked list

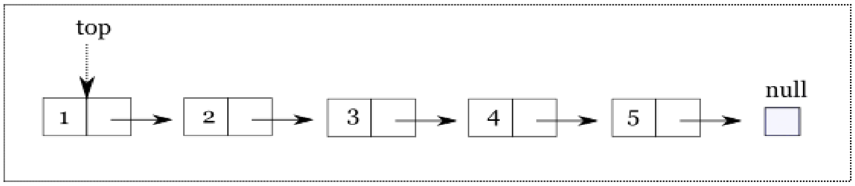
Declaration

```
class node
{
public:
    int value;
    node* next = nullptr;

    node(int value);
};


class stack
{
    node* top = nullptr;

public:
    void push(int value);
    int pop();
    int peek();
    bool is_empty();
};
```



Implementation

Node constructor

```
node::node(int value)
{
    this->value = value;
}
```

Push operation

```
void stack::push(int value)
{
    node* tmp = new node(value);
    tmp->next = top;
    top = tmp;
}
```

Pop operation

```
int stack::pop()
{
    if (is_empty())
    {
        std::cout << "stack is empty, can not pop\n";
        return -1;
    }
    else
    {
        int tmp = top->value;
        top = top->next;

        return tmp;
    }
}
```

Peek operation

```
int stack::peek()
```

```
{
    if (is_empty())
    {
        std::cout << "stack is empty, can not pop\n";
        return -1;
    }
    else
    {
        return top->value;
    }
}
```

Function is_empty()

```
bool stack::is_empty()
{
    return (top == nullptr);
}
```

Performance

Operation	Complexity
push	O(1)
pop	O(1)
peek	O(1)

Pros

- Operations takes constant time
- Unlimited size

Cons

- Extra space and time to deal with references

Application