Data Structures:

A data structure may be static or dynamic. A static data structure has a fixed size. This meaning is different from the meaning of static modifier. Arrays are static; once we define the number of elements it can hold, the number doesn't change. A dynamic data structure grows and shrinks at execution time as required by its contents. A dynamic data structure is implemented using links.

Data structures may further be categorized into linear data structures and non-linear data structures. In linear data structures every component has a unique predecessor and successor, except first and last elements, whereas in case of non-linear data structures, no such restriction is there as elements may be arranged in any desired fashion restricted by the way we use to represent such types.

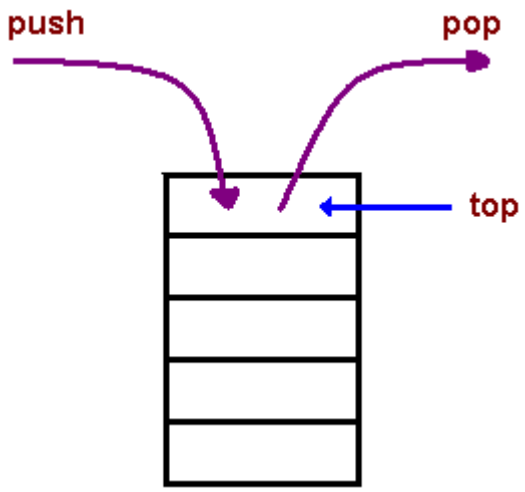
Abstract Data Type:

ADT may be defined as a set of data values and associated operations that are precisely specified independent of any particular implementation. Thus an Abstract Data Type is an organized collection of information and a set of operations used to manage that information.

The basic difference between abstract data type (ADT) and concrete data type is that the latter allow us to look at the concrete representation, whereas the former hide the representation from us.

Stacks

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

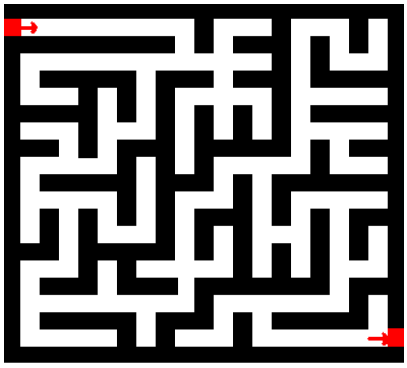


A stack is a **recursive** data structure. Here is a structural definition of a Stack:

a stack is either empty or it consists of a top and the rest which is a stack;

Applications

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.



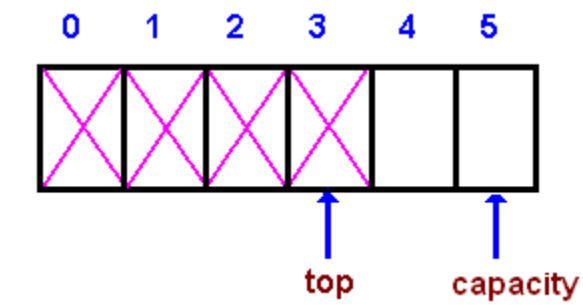
Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

- Language processing:
  - Space for parameters and local variables is created internally using a stack.
  - Compiler's syntax check for matching braces is implemented by using stack.
  - support for recursion

Implementation

Array-based implementation



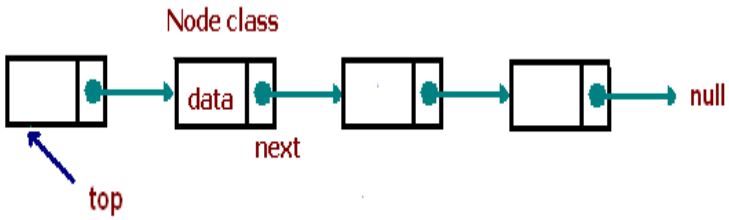
In an array-based implementation we maintain the following fields: an array A of a default size (≥ 1), the variable top that refers to the top element in the stack and the capacity that refers to the array size. The variable top changes from -1 to `capacity - 1`. We say that a stack is empty when `top = -1`, and the stack is full when `top = capacity-1`.

In a fixed-size stack abstraction, the capacity stays unchanged, therefore when top reaches capacity, the stack object throws an exception.

In a dynamic stack abstraction when top reaches capacity, we double up the stack size.

Linked List-based implementation

Linked-List based implementation provides the best (from the efficiency point of view) dynamic stack implementation.




```
#include<stdio.h>
#define SIZE 5            /* Size of Stack */
int s[SIZE], top = -1; /* Global declarations */

int Sfull()
{
        /* Function to Check Stack Full */
        if (top == SIZE - 1)
                return 1;
        return 0;
}
int Sempty()
{
        /* Function to Check Stack Empty */
        if (top == -1)
                return 1;
        return 0;
}
void push(int elem)
{
   /* Function for PUSH operation */
    if (Sfull())
     printf("\nOverflow!!!!\n");
    else
          {
     ++top;
     s[top] = elem;
    }
}
```

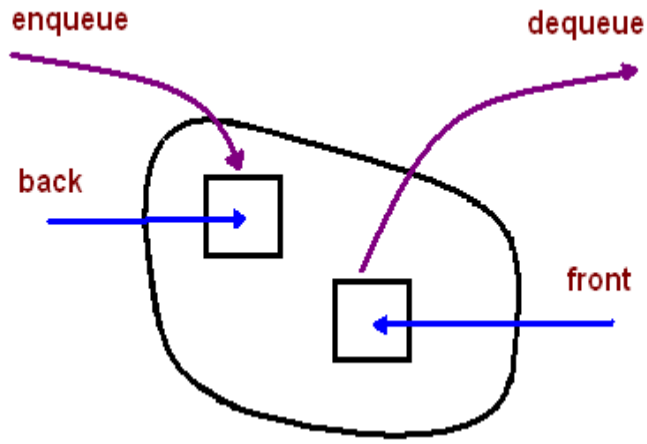
```
int pop()
{
        /* Function for POP operation */
    int elem;
    if (Sempty())
    {
     printf("\nUnderflow!!!!\n");
     return (-1);
    }
    else
    {
     elem = s[top];
     top--;
     return (elem);
    }
}
void display() { /* Function to display status of Stack */
          int i;
          if (Sempty())
           printf("Empty Stack\n");
          else
          {
                  for (i = 0; i <= top; i++)
                          printf("%d\n", s[i]);
                  printf("^Top");
          }
}
int main() {
/* Main Program */
 int opn, elem;
 do {
  printf("\n ### Stack Operations ### \n\n");
  printf("\n Press 1-Push, 2-Pop,3-Display,4-Exit\n");
  printf("\n Your option ? ");
  scanf("%d", &opn);
  switch (opn) {
  case 1:
   printf("\nRead the element to be pushed ?");
   scanf("%d", &elem);
   push(elem);
   break;
```

```
 case 2:
  elem = pop();
  if (elem != -1)
   printf("\nPopped Element is %d \n", elem);
  break;
 case 3:
  printf("\nStatus of Stack\n");
  display();
  break;
 case 4:
  printf("\nTerminating \n");
  break;
 default:
  printf("\nInvalid Option !!! Try Again !! \n");
  break;
 }
 printf("\nPress a Key to Continue . . . ");
} while (opn != 4);
}
```

Queues

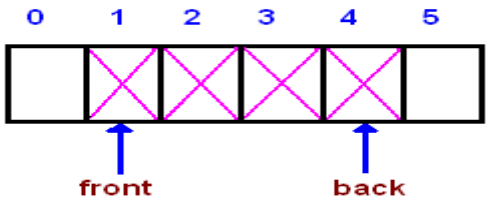
A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed **enqueue** and **dequeue**. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

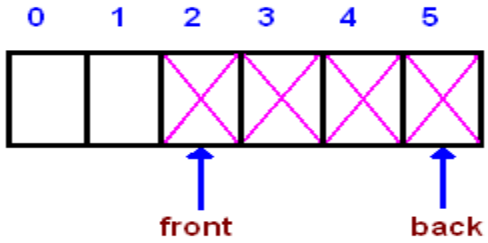


Circular Queue

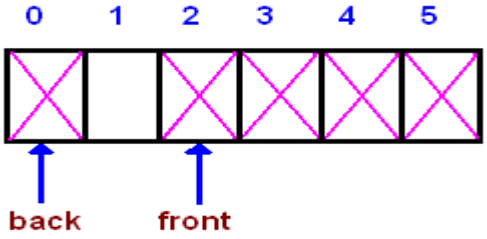
Given an array A of a default size (≥ 1) with two references *back* and *front*, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index. Here is a picture that illustrates the model after a few steps:



As you see from the picture, the queue logically moves in the array from left to right. After several moves *back* reaches the end, leaving no space for adding new elements



However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until *front*. Such a model is called a **wrap around queue** or a **circular queue**



Finally, when *back* reaches *front*, the queue is full. There are two choices to handle a full queue:a) throw an exception; b) double the array size.

The circular queue implementation is done by using the modulo operator (denoted %), which is computed by taking the remainder of division (for example, 8%5 is 3). By using the modulo operator, we can view the queue as a circular array, where the "wrapped around" can be simulated as "back % array_size". In addition to the back and front indexes, we maintain another index: *cur* - for counting the number of elements in a queue. Having this index simplifies a logic of implementation.