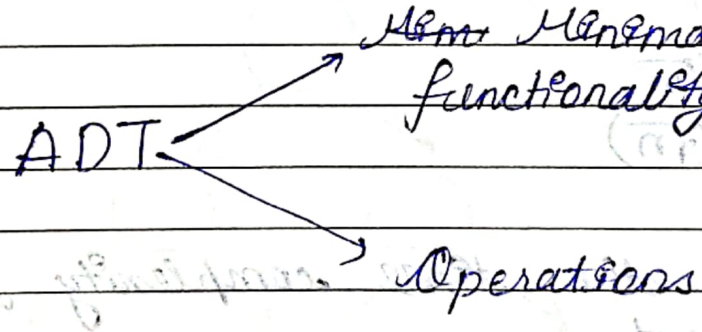
Arrays & Abstract Data types

ADT's or abstract data types are the ways of classifying data structures by providing a minimal expected interface & some set of methods. It is very similar to when we make a blueprint before actually getting into doing some job, be it constructing a computer or a building. The ~~tool~~ blueprint comprises all the minimum req^d logistics & the roadmap to pursuing the job.



Array - ADT

An array ADT holds the collection of given elements (can be int, float, custom) accessible by their index.

1) Minimal req^{d} functionality

We have two basic functionalities of an array, a get function to retrieve the element at index i & set function to assign the element to some index in the array.

- get(i) - get element i
- set(i, num) - set element i to num

2) Operations

We can have a whole lot of different ~~ope~~ operations on the array we created, but we'll limit ourselves to some basic ones.

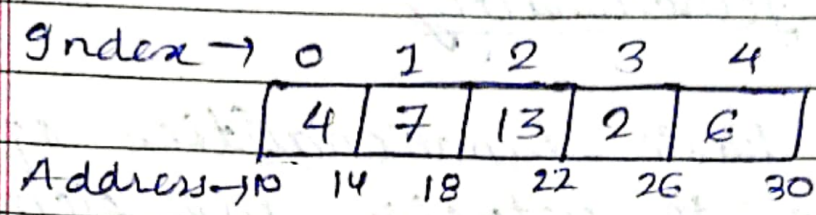
- Max()
- Min()
- Search(num)
- Insert(i, num)
- Append(x)

Static & Dynamic Arrays

→ Static Arrays - size cannot be changed

→ Dynamic Arrays - size can be changed

Memory representations of array



→ Elements are stored in contiguous memory locations

→ Elements in an array can be accessed using the base address in const. time ⇒ → O(1)

→ Although changing the size of an array is not possible, one can always reallocate it to some bigger memory location. Therefore resizing an array is a costly operation

An Abstract data type is just another data type such as int or float, with some user defined methods & operations. It's a kind of customized data type.

Suppose we want to build an array as a ADT with our customized set of values & customized set of operations in a heap. Let's name this customized array myArray.

Let our set of values which will represent our customized array include these parameters.

→ total_size

→ used_size

→ base_address

And the operations include operators namely

→ max()
→ get(i)
→ set(i,num)
→ add(another_array)

So now when we are done creating a blueprint of the customized array. We can very easily code their implementation.

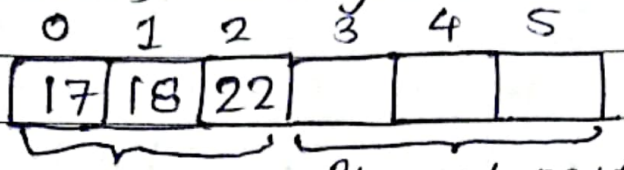
Understanding the ADT above:

(1) total_size: This stores the total reserved size of the array in memory location.

(2) used_size:
This stores the total reserved size of the memory location used.

(3) base_address:
This is a pointer that stores the address of the first element of the array.

For e.g.



Used part of location — Unused part for later use

Understanding the snippet below

→ First, we will define a structure.
→ C is a structure used to define customized data types
→ Keep the ~~tito~~ blueprint we made in the last tutorial by your side. Define the structure elements, integer variables total_size & used_size, & an integer pointer to point at the address of the first element.
→ We are now ready with our customized data type. Let's define some functions which will feature

- Creating an array of this data type
- Printing the contents of this array
- Setting values in this array.

~~Creating~~ Create a void function ~~a~~ create Array by passing the address of a struct data type a, & integers tSize & uSize. We can very ~~easily~~ easily assign the tSize & uSize given from the main, to the total_size & used_size

of the struct myArray a. Use either of the methods given below.

```
(*a).total_size=tSize;
        or
a->total_size=tSize;
```

Code snippet 1: Syntax for assigning structure elements to structure pointers

Similarly, assign the integer pointer ptr, the address of the reserved memory location using malloc. Do use header file <stdlib.h> for malloc.

```
a->ptr=(int *)malloc(tSize * sizeof(int));
```

Snippet 2: Using malloc

→ We will now create a show function to display all the elements of the struct myArray. We will simply pass the address of the struct myArray a. To print all the elements, we will traverse through the whole struct & print each struct element till the iterator reaches the last element. We will use a->used_size to define the loop size. Use (a->ptr)[i] to access each element.

→ We will now create a setVal func^n to set all values to this struct myArray a & pass the address of the same. Use scanf to assign values to each element via (a→ptr)[i].

Code:

```
#include<stdio.h>
#include<stdlib.h>

struct myArray
{
  int total_size;
  int used_size;
  int *ptr;
};

void createArray(struct myArray *a, int tSize,
                 int uSize)
    {
  // (*a).total_size = tSize;
  // (*a).used_size = uSize;
  // (*a).ptr = (int*) malloc(tSize * sizeof(int));

  a→total_size = tSize;
  a→used_size = uSize;
  a→ptr = (int*)malloc(tSize * sizeof(int));

  }
```

```
void show(struct mydrray *a)
{
 for(int i=0; i<a->used_size; i++)
  {
    printf("%d\n",(a->ptr)[i]);
  }
}

void setVal(struct mydrray *a)
 { int n;

   for(int i=0; i<a->used_size; i++)
    {
   printf("Enter element %d", i);
   scanf("%d", &n);
   (a->ptr)[i]=n;
     }
 }

int main(){
 struct mydrray marks;
 createdrray(&marks, 10, 2);
 printf("We are running setVal now\n");
  setVal(&marks);

 printf("We are running show now\n");
 show(&marks);

 return 0;
}
```

o/p of program

```
We are running setVal now
Enter element 0:12
Enter element 1:13

We are running show now
12
13
```

Operations on Arrays in Data Structures

While there are many operations that can be implemented & studied, we only need to be familiar with the primary ones at this point. An array supports the foll. operations:

- Traversal
- Insertion
- Deletion
- Search

TRAVERSAL

Visiting every element of an array once is known as traversing the array.

For use cases like:

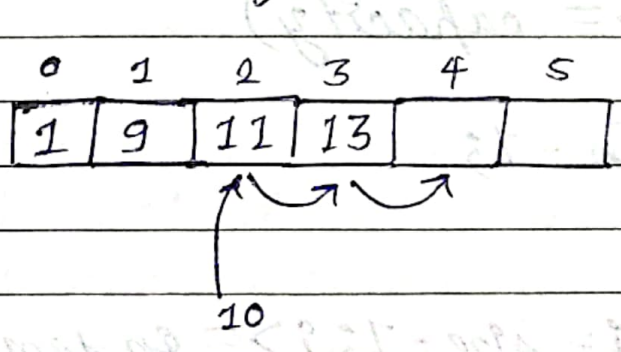
- Storing all elements - Using scanf()
- Printing all elements - Using printf()
- Updating elements

An array can be easily ~~or~~ traversed using for loop in C.

Note: We cannot exceed the size limit of an array.

INSERTION

An element can be inserted in an array at a specific position. For this operation to succeed, the array must have enough capacity. Suppose we want to add an element ~~at~~ 10 at index 2 in the array, then the elements after index 1 must get shifted to their adjacent right to make ~~their~~ way for a new element.



When no pos" is specified, it's best to insert the element at the end to avoid shifting, & this is when we achieve best runtime O(1).

Code for Insertion:

```
#include <stdio.h>

void display(int arr[], int n)
{
  // code for Traversal
      for(int i=0; i<n; i++)
      {
          printf("%d", arr[i]);
      }
  printf("\n");
}

int indInsertion(int arr[], int size,
          int element, int capacity, int index)
{  // code for Insertion

  if (size >= capacity)
     {
       return -1;
     }

for (int i = size-1; i >= index; i--)
   {
     arr[i+1] = arr[i];
   }
   arr[index] = element;
   return 1;
}
```

```
int main(){
    int arr[100]={7,8,12,27,88};
    int size=5, element=45, index=1;
    display(arr,size);
    indInsertion(arr,size,element,100,index);
    size+=1;
    display(arr,size);
    return 0;
}
```

o/p

7 8 12 27 88

7 45 8 12 27 88

1) We will start by declaring an array length 100. Initialize this array with some 4-5 elements. This will be our used memory.

2) We'll create a void display function using the method of traversal. Pass this array to the display function by value or by reference. And print the elements. ~~Printing the elements of an array has also already been covered~~

3) We'll now create an integer function ind Insertion (integer, just to check if the operation succeeds). Before that, create an

Integer variable size to store the used size of the array. Pass into this void function the array and its used size, the element to be inserted & the total size, & index where its inserted.

4) In the indInsertion function, write the case of validity. Here, we'll check if the index is within the range [0, 100]. We'll continue if it's valid; otherwise return -1.

5) Create a for loop to shift the elements from ~~the~~ the index to the last elements to their adjacent right. This way, we'll create a void at the index we want to insert in.

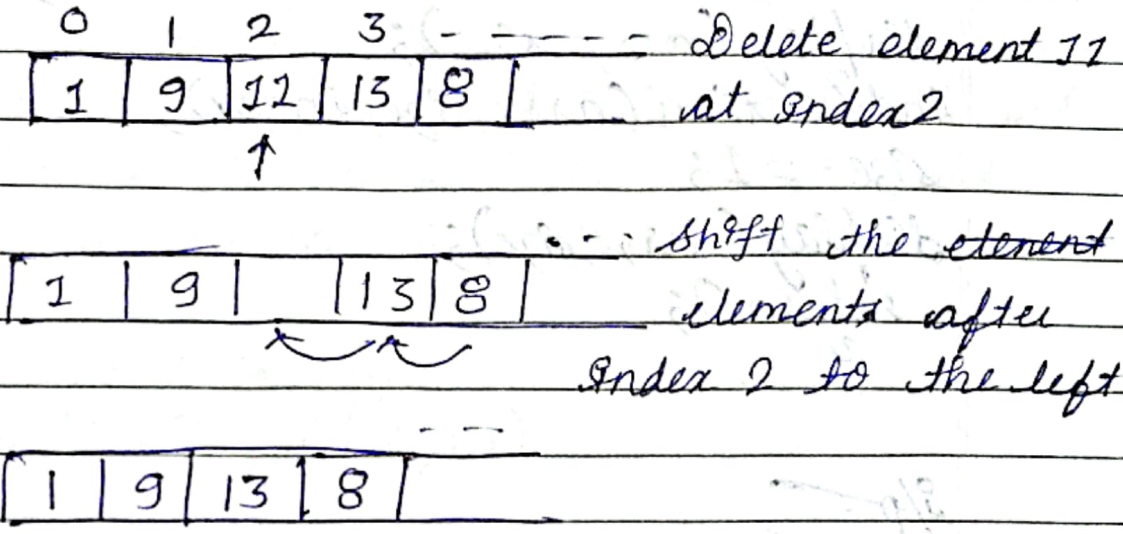
6) Insert the element in the index. Return 1 on completion.

DELETION

~~#include <stdio.h>~~

An element at a specified pos^{n} can be deleted, creating a void that needs to be ~~shifted~~ fixed by shifting all the elements to their adjacent left, as illustrated in the figure below.

We can also bring the last element of the array to fill the void of order isn't important



Code:

```
#include <stdio.h>

void display(int arr[], int n)
{
    // Code for Traversal
    for(int i=0; i<n; i++)
    {
        printf("%d", arr[i]);
    }
    printf("\n");
}

void indDeletion(int arr[], int size, int index)
{
    // Code for Deletion
    for(int i=index; i<size-1; i++)
    {
        arr[i] = arr[i+1];
    }
}
```

```
int main()
{
 int arr[100]={7,8,12,27,88};
int size=5, element=45, index=0;
display(arr, size);
indDeletion(arr, size, index);
size--=1;
display(arr, size);
 return 0;
}
```

i/p :-

o/p

7 8 12 27 88

8 12 27 88