ASSIGNMENT-5

M. Udaykumar
192321073

1. i) write the algorithm for insertion sort and sort the following sequence: 3, 1, 4, 1, 5, 9, 2, 6, 5.

ii) Explain the procedure for merge sort and perform merge sort for the following inputs. Also, shows the result for each step of iteration. 64, 8, 216, 512, 27, 729, 0, 1, 343, 125.

i) Algorithm

```
Insertion sort (a[n]);
for(i=1; i<n-1; i++)
  key = a[i]
   j = i-1
  while j>0 and a[j]>key;
      a[j+1] = a[j]
        j = j-1
      a[j+1] = key
```

Sorting

Initialize temp variable

temp

	3	1	4	1	5	9	2	6	5

Step 1:

a[0]=3 , a[1]=1

a[0] > a[1]

1 goes to temp

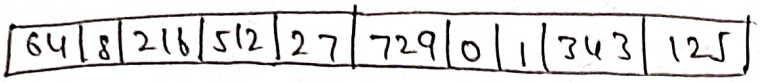
1	3	4	1	5	9	2	6	5

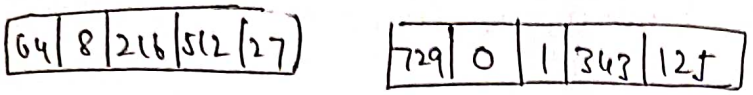
1<3, a[0]=1

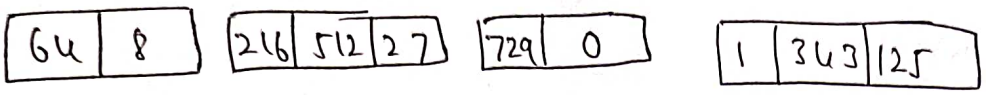
	1	3	4	1	5	9	2	6	5

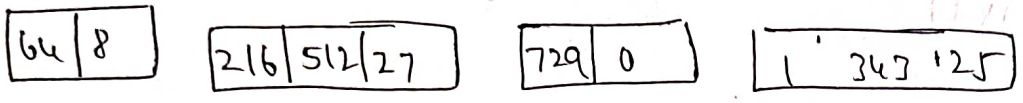
ii) Merge sort

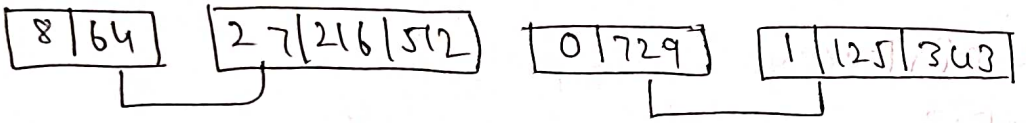
Initial array: [64, 8, 216, 512, 27, 729, 0, 1, 343, 125]

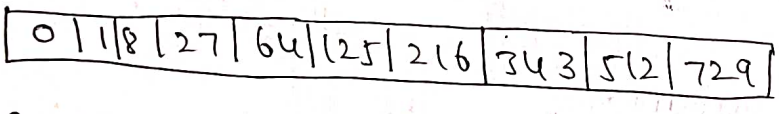












2. Draw the concept map of partioning in quick sort, by to write an algorithm for it which it as follow

Develop a program considering these steps

Step-1. Choose the height index value has pivot

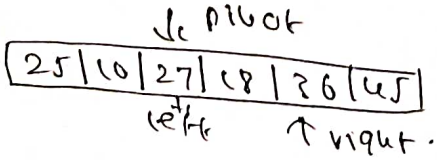
Step-2. take two variable to point left and right of the list excluding pivot.

Step 3: left points to the low index

using elements your own.

a[left]=36, a[pivot]=a[right]=27, a[pivot] < a[left],

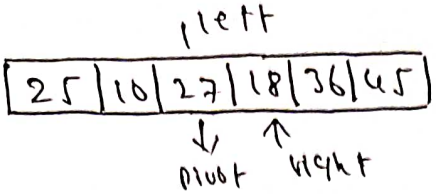
so swap



a[left] = a[pivot] = 27, a[right] = 36

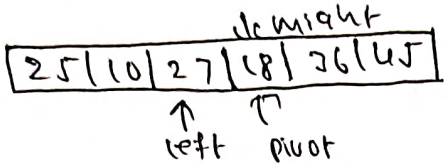
→ since pivot is at left, so algorithm starts from right and move to left.

a[pivot] < a[right], right moves one position forward.



a[left] = a[pivot] = 27, a[right] = 18

a[pivot] > a[right] so swap



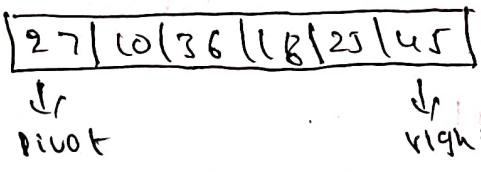
→ since pivot is at right, so algorithm starts from left and moves to right.

a[left] = 18 a[pivot] = a[right] = 27

a[pivot] > a[left], so left moves one position forward

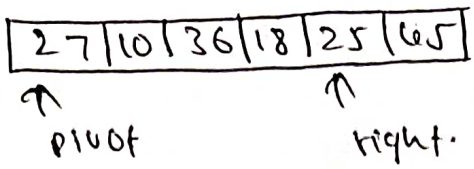
→ now, a[pivot], a[left] and [right] are same, so than are pointing the same element, it represents the termination of procedure.

a[ ] = { 27, 10, 36, 18, 25, 45 }



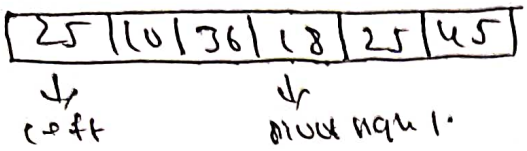
Compare a[pivot] & a[right]

a[pivot] < a[right], so right moves forward one position.



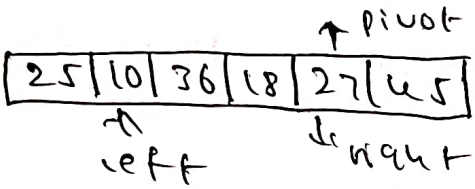
a[left] = a[pivot] = 27, a[right] = 25

a[pivot] > a[right], so swap



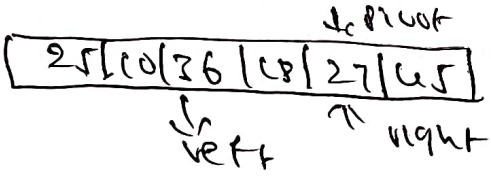
→ since, pivot is of right, so algorithm starts from left and moves to right

a[pivot] > a[left], so also moves one position to right



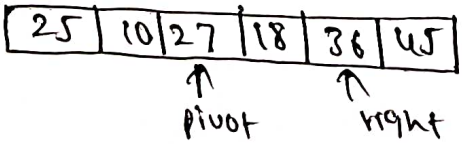
a[left] = 10, a[pivot] = a[right] = 27

a[left] < a[pivot], so left moves forward



a[left] = 36, a[pivot] = a[right] = 27, a[pivot] < a[left]

so, swap



a[left] = a[pivot] = 27, a[right] = 36

* since, pivot in at left, so algorithm starts from right and move to left

a[pivot] < a[right] right moves one position forward

25	10	27	18	36	45

a[left] = 18 a[pivot] = a[right] = 27

a[pivot] > a[left], so left moves one position forward

* Elements that are right side of element 27 greater than if and the elements that are left side of 27 are smaller

25	10	18	27	36	45

left subarray Right sub array.

* Now in a similar manner, quick sort algorithm is seperately applied to the left and right subarray.

10	18	25	27	36	45

CSA0389

10/10

NAME :- M. Uday Kumar

REG NO :- 192325073

COURSE :- Data Structures for stack overflow

COURSE CODE :- CSA0389

Dept :- AI & ML

Date :- 31/07/2024.

M. Uday Kumar
192325073

Describe the concept of Abstract data type (ADT) and how they differ from concrete data structures. Design an ADT for a stack and implement it using arrays and linked list in C. Include operations like Push, Pop, Peek, is Empty, is Full and peek.

Sol:- **Abstract Data Type (ADT)**

An Abstract Data Type (ADT) is a theoretical model that defines a set of operations and the semantics (behaviour) of those operations on a data structures, without specifying how the data structures should be implemented. It provides a high level description of what operations can be performed on the data and what constraints apply to those operations.

Characteristics of ADTs:-

Operations: Defines a set of operations that can be performed on the data structure.

Semantics: Specifies the behaviour of each operations.

Encapsulation: Hides the implementation details focusing on the interface provided to the user.

ADT For Stack:-

A stack is a fundamental data structures that follows the last In, First out (LIFO) principle. It supports the following operations.

Push Adds an element to the top of the stack.

Pop: Adds an element to the top of the stack.

Peek: Returns the element from the top of the stack without removing it.

isEmpty:- Checks if the stack is empty.

isFull:- Checks if the stack is full.

Concrete Data structures:

* The implementations using arrays and linked list are specific ways of implementing the stack ADT in C.

How ADT differ from concrete Data structures:

ADT focuses on the operations and their behaviour, while concrete data structure focus on how those operations are realized using specific programming constructs (arrays are linked lists).

Advantages of ADT:

By separating the ADT from its implementation, achieve modularity, encapsulation and flexibility in designing and using data structures in programs. This separation allows for easier maintenance, code reuse, and abstraction of the complex operations.

Implementation in C using Array:

```
#include <stdio.h>
#define MAXSIZE 100
type def struct {
  int items[MAX_SIZE];
  int top;
} stack Array;
int main() {
    StackArray stack;
    stack.top = -1;
    stack.items[++stack.top] = 10;
    stack.items[++stack.top] = 20;
    stack.items[++stack.top] = 30;
if (stack.top != -1) {
    printf("top element: %d\n", stack.items[stack.top]);
```

```
} else {
Printf ("stack is empty : \n");
}
if (stack.top != -1) {
 Printf ("Popped elements : %d\n", stack.items [stack.top--]);
}else {
  Printf ("stack underflow: \n");
}
if stack.top != -1) {
 Printf (" Popped elements: %d \n", stack.items [stack.top--]);
 } else{
 Printf(" stack underflow :\n");
 }
if (stack.top != -1) {
   Printf ("Top element after pops: %d \n", stack.items stack.top);
 }else{
    Printf("stack is empty:\n");
  }
  return 0;
 }
```

Implementation in C using linked list:-

```
#include <stdio.h>
#include <stdio.h>
type def struct node {
      int data;
   struct node * next;
} node;
int main () {
  node * top = NULL;
```

```
Node* newNode = (Node*) malloc (sizeof (Node));
if (newnode == NULL) {
Printf ("memory allocation failed:\n");
return 1;
}
new node -> data=10;
new node -> next = top;
top = newnode;
newnode = (Node*) malloc (size of (Node));
if (newnode == NULL) {
   Printf("memory allocation failed:\n");
   return 1;
}
new node -> data==0;
new node -> next = top;
top = new node;
new node = (node*) malloc (sizeof (Nodes));
if (new node == NULL) {
Print f("memory allocation failed:\n");
return 1;
}
new node -> data=30;
new node -> next = top;
top = newNode;
if (top != NULL) {
 Print f ("Top element: %d \n") top -> data);
}else{
Print f ("stack is empty:\n");
}
```

```
if (top != NULL) {
node *temp = top;
printf("Popped element: %d\n", temp->data);
top = top->next;
free(temp);
} else {
printf("stack underflow:\n");}
if (top != NULL) {
printf("Top element after pops: %d\n", top->data);
} else {
printf("stack is empty!\n");
}
while (top != NULL) {
node *temp = top;
top = top->next;
free(temp);
}
return 0;
}
```

The university announced the selected candidates register number for placement training. The structure xxx.reg no 2016 2010 wishes to check whether his name is listed or not. The list is not sorted in any order. Identify the searching technique that can be applied and explain the searching steps with the suitable procedure. List includes 2016 2015, 2014 2011 2014, 2017, 2016 2010, 2014 2056, 2016 2003.

linear search:-

linear search works by checking each element in the list one by one until the desired element is found or end of the list is reached. It's a simple searching technique that doesn't require any prior sorting of the data.

steps for linear search:-

1. Start from the first element.
2. check if the current element is equal to the target element.
3. If the current element is not the target move to the next element in the list.
4. continue this process until either the target element is found or you reach the end of the list.
5. If the target is found, return its position. If the end of the list is reached and the element has not been found, indicate that element is not present.

Procedure:-

Given the list:

20142015, 20142033, 20142011, 20142017, 20142010, 20142051, 20142002

1. Start at the first element of the first.
2. compare '20142010' with '20142015' (first element), 20142033 (second element), '20142011' (third element), '20142017' (fourth element) these are not equal.
3. compare '20142010' with '20142010' (fifth element). They are equal.
4. The element '20142010' is found at the fifth position (index in the list.

C code for linear search:-

```
#include <stdio.h>
int main() {
 int regnumbers[] = {20162015, 20162033, 20162016, 20162017,
                     20162010, 20162056, 20162003};
 int target = 20162010;
 int n = sizeof(regnumbers)/sizeof(regnumbers[0]);
 int found=0;
 int i;
 for(i=0; i<n; i++) {
  if(regnumbers[i]==target) {
  printf("Registration number %d found at index %d\n",target,i);
   found=1;
   break;
  }
 }
 if(!found) {
  printf("Registration number %d not found in list.\n",target);
 }
  return 0;
}
```

Explanation of the code:-

1. The 'regnumbers' array contains the list of registration numbers.
2. target is the registration numbers we are searching for.
3. 'n' is the total number of elements in array.
4. Iterate through each element of the array.

5. If the current element matches the 'target', print its index and set the 'found' flag to '1'.

6) If the loop completes without finding the target, print.

7) The program will print the index of the found registration number or indicate that the registration is not present.

Output:- Registration number 2014010 found at index 4.

3. write pseudocode for stack operations.

1. Initialize stack():

Initialize necessary variable or structure to represent the stack.

2. Push (elements):

if stack is full:

Print "stack overflow"

else:

add element to the top of the stack

increment top pointer

3. POP():

if stack is empty:

Print ("stack underflow")

return null (or appropriate error value)

else:

remove and return element from the top of the stack

decrement end pointer

4. peek():
if stack is empty:
print "stack is empty".
return null (or appropriate error value)
else:
return element at the top of the stack (without removing it)

5) isEmpty():
return true if top is -1 (stack is empty)
otherwise, return false

6) isFull:
return true, if top is equal to maxsize -1 (stack is full)
otherwise, return false.

Explanation of the pseudocode:

* Initializes the necessary variables or data structures to represent a stack.
* Adds an element to the top of the stack. Checks if the stack is full before pushing.
* Removes and returns the element from the top of the stack. Checks if the stack is empty before popping.
* Returns the element at the top of the stack without removing it. checks if the stack is empty befor peeking.
* checks if the stack is empty by inspecting the top pointer or equivalent variables.
* Checks if the stack is full by comparing the top pointer or equivalent variable to the maximum size of the stack