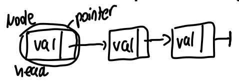
Linked Lists:
A single linked list is a list made up of nodes that consists of two parts (data and link)



```
struct Node { //struct is used to define a data type
int data; //is the value saved in the memory of the list
struct Node* next; //stores the address of the next node in the list, the list is linked (it is a pointer to the next node)
} Node * head; //points to the head node (so it is not the head node)
```

Each node contains 2 cells: an integer (value) and an address of type Node*.
The first node is called head node.
The address of the head node gives access to the complete list (to all its elements).
The address of the last array is null.

To create a new node:
-create a new node with malloc (it creates a new memory block of size of the variable)
-assign a value to that node
-the node after the created node is null

Formally:

```
struct Node *newNode = malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = NULL;
```

To insert an element into a linked list:
-first traverse all the elements and then put at the end --> Cost of O(n)
Deleting a node at nth position:
-if we want to remove the node at nth position we first have to link the n-1 node with the n+1, so the list remains linked
-then we can free space

```
void Delete(int n) {
struct Node* temp1 = head;
if (n==1) {
head = temp1->next; //head now points to second node
free(temp1);
return;
int i;
for (i = 0; i<n-2; i++) {
temp1 = temp1 -> next;
//temp1 points to (n-1)th Node
struct Node* temp2 = temp1-> next; //nth node
temp1-> next = temp2->next; //n+1th node
free(temp2); //delete temp2
int main() {
head = NULL;
InsertList(2); //Insertlist inserts the element at the end of the list
InsertList(4);
InsertList(6);
InsertList(5); //List = 2,4,6,5
printf("Enter a position:")
scanf("%d", &n);
Delete(n)
}
```

Double Linked lists:
-each node has two links (address of previous node and address of next node) and 1 value--> total of 3 cells per node
-allows a reverse look-up (from a pointer we can see the address of the previous

and next nodes)
-Disadvantage: requires extra memory for pointer to previous node
Implementation in C:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
int data;
struct Node* next;
struct Node* prev;
}
struct Node* head; //global variable (accessible everywhere) is a  pointer to head
node

struct Node* GetNewNode(int x) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)) //local newnode to
function getnewnode
newNode->data = x;
newNode->prev = NULL;
newNode -> next = NULL;
return &NewNode;

void InsertAtHead(int x) {
struct Node* newNode = GetNewNode; //local newnode to function insertathead
if (head == NULL) {
head = newNode;
return;
}
head->prev = newNode;
newNode->next = head;
head = newNode;
int main() {
```

QuickSort:
-Time Complexity: O(log(n)) in average case and O(n^2) in worst case
-In-place algorithm: takes almost constant space (doesn't need extra memory)
-the element taken as reference is called pivot
-all the elements greater than the pivot are right to the pivot
-all the elments smaller than the pivot are left to the pivot
-It is a divide and conquer algorithm
-the final index i+1 (lomuto) and i (hoare) is the position where the index should go
we have to sort the sub-arrays as well till we reach single elements (kind of like base cases where we stop the recursion (l = r: so l < r not true anymore (the program stops when start >= end), the recursive call stop for that subarray)
After dividing the array in two parts, first we solve the left array recursively until no further division, and only then we start to sort the right subarray
Lomuto Partition:
-takes last element as pivot
-starts from first element and confronts it with pivot
-if element is less (or bigger) than pivot we swap it with the next element, and now we consider the i+1 element and repeat the same procedure
-if it is ascending order and element is bigger we go to next element (no need to swap) and as soon we find one bigger we memorize it and we swap it with the next one
-after 1 iteration the smallest (or biggest) will be sorted
-the pivot is now the element at index pivot - 1
-repeat same procedure till array is sorted

-order of the array (ascending or descending) is determinate by this condition: if (A[j] <= pivot) (in this case it's ascending order)
-first sorts left array (from left to middle-1) and then right sub-array (from middle + 1 to right) -->middle element is pivot and is already at right position

Stacks:
-push() operation: inserts data onto stack
-pop() operation: deletes last inserted element from the stack (top elemenet of stack)
-top(): returns the last inserted element without removing it
-size(): returns the number of elements in the stack
-isEmpty(): returns true if stack is empty, else returns false
-isFull(): returns true if stack is full (same lements after a number of operations), else returns false