Shourya Gupta
19095089
CSE-102
Electronics Engineering

1(a).
We can use Hashing to implement python Dictionary in C.

Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value

Pseudo-Code
```
void hashing(key,value){
 arr[32]  //create array of required size
index=hash_function(key)   //→ will return an index less than 32
insert(index)        //→ will insert the value at that index one index can store more than one value

}
//to get value from key
get_Value(key){
index=hash_function(key)   //→ will return an index less than 32
get(index)  //→ return value
}
```

1(b).
I think the book index present at the end of the book is a like Python Dictionary having key value pairs which can implement using hashing in C.
So according to me Hashing with direct chaining is the best way to do so, pseudo_code for which will be same as in 1(a).

2(a).
The Dijkstra's Algorithm works on a weighted graph with non-negative edge weights and gives a Shortest Path Tree. It is a greedy algorithm, which sort of mimics the working of breadth first search and depth first search.
We will need priority Queue data structure to implement Dijkstra's algorithm in C.

The algorithm iterates once for every vertex in the graph; however, the order that we iterate over the vertices is controlled by a priority queue. The value that is used to determine the order of the objects in the priority queue is the distance from our starting vertex. By using a priority queue, we ensure that as we explore one vertex after another, we are always exploring the one with the smallest distance.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Whenever distance of a vertex is reduced, we add one more instance of vertex in priority_queue. Even if there are multiple instances, we only consider the instance with minimum distance and ignore other instances.

Pseudo-Code

```
Dijsktra(G, S)
   D(S) = 0
   Q = G(V)

    while (Q != NULL)
        u = extractMin(Q)
        for all V in adjacencyList[u]
          if (D(u) + weight of edge < D(V))
                D(V) = D(u) + weight of edge
                decreasePriority(Q, V)
```

2(b).
Yes, we can can use priority queue to implement A* algorithm in C.

Each time A* enters a node, it calculates the cost, f(n)(n being the neighboring node), to travel to all of the neighboring nodes, and then enters the node with the lowest value of f(n).

Implementation
So, we start from the source node, and then extract all the neighbors, then for each neighbor add its g value (distance from parent to this neighbor) to the g value of the parent, then add its h value to this, giving you the f value (f = g + h). Then add this all these neighbors to a priority queue with f values. These nodes are now open to calculate and the source node is closed to calculate. Now we go to the node with the shortest f value (if the destination is not already reached) and then extract it only if this node does not already has a shorter path available (shorter f value, which is maintained in a list for each node, which is initially infinity for all). If there is already a shorter node to a current node, this means that this particular path is not the shortest and hence we do not expand its neighbors and we can make this node as closed and return to the next shortest path in the priority queue. This is now repeated till we reach our destination. the moment we reach the destination it is the shortest path, and a guaranteed one if the heuristic is consistent.
Here heuristic means the estimated value to reach to goal node from source node.

Pseudo-Code:

```
make an openlist containing only the starting node
   make an empty closed list
   while (the destination node has not been reached):
       consider the node with the lowest f score in the open list
       if (this node is our destination node) :
           we are finished
       if not:
          put the current node in the closed list and look at all of its neighbors
          for (each neighbor of the current node):
           if neighbor has lower g value than current and is in the closed list
                   replace the neighbor with the new, lower, g value
                   current node is now the neighbor's parent
           else if current g value is lower and this neighbor is in the open list:
                   replace the neighbor with the new, lower, g value
                   change the neighbor's parent to our current node

           else if this neighbor is not in both lists
                   add it to the open list and set its g
```

We can use any data structure to implement open list and closed list but for best performance we use priority queue