Saurabh singh yadav
19095087
CSE-102
Electronics Engineering

1(a).
We can use Hashing to implement python Dictionary in C.
We can use ***Hashing with Direct Chaining***.

In hashing, large keys are converted into small keys by using ***hash functions***. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key . By using that key you can access the element in O(1) time. Using the key, the hash function computes an index that suggests where an entry can be found or inserted.

Pseudo-Code
```
void hashing(key,value){
 arr[16]  //create array of required size let say
index=hash_function(key)   //→ will return an index less than 16
insert(index)        //→ will insert the value at that index one index can store more than one value

}

get_Value(key){
index=hash_function(key)   //→ will return an index less than 16
get(index)  //→ return value
}
```

1(b).
I think the book index present at the end book is a like Python Dictionary 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).
Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph.
We will need queue data structure to implement Dijkstra's algorithm in C.

A special type of queue known as ***priority Queue*** is used in Dijkstra algorithm.
We overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.
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

```
Dijkstra(G, S)  //here G is graph and S is Source
  for each vertex V in G
     distance[V] = infinite
     previous[V] =NULL       //previous is maintained to get path
     If V != S, add V to Priority Queue Q
  distance[S] =0

  while Q IS ! EMPTY
     U       // Extract MIN from Q
     for each unvisited neighbour  V of U
        temp  // distance[U] + edge_weight(U, V)
        if temp < distance[V]
           distance[V] =temp
           previous[V] = U
  return distance[], previous[]
```

2(b).
Yes, we can can use ***priority queue*** to implement A* algorithm in C.
Implementations of A* use a ***priority queue*** to perform the repeated selection of minimum cost nodes to expand.

A* is based on using heuristic methods to achieve *optimality* and *completeness*, and is a variant of the best-first algorithm.

When a search algorithm has the property of optimality, it means it is guaranteed to find the best possible solution, in our case the shortest path to the finish state. When a search algorithm has the property of completeness, it means that if a solution to a given problem exists, the algorithm is guaranteed to find it.

Each time A* enters a state, 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)`.

These values are calculated with the following formula: f(n)=g(n)+h(n)

`g(n)` being the value of the shortest path from the start node to node *n*, and `h(n)` being a heuristic approximation of the node's value.

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:
Add start node to list
for all the neighboring nodes,find the least cost F node
switch to the closed list
  for all nodes adjacent to the current node
  if the node is not reachable ,ignore it. Else
      if the node is not on the open list,move it to the open and calculate f,g,h.
      If the node is on the open list,check if the path it offers is less than the current path and change to it does so.
Stop working then
    You find the destination
    You cannot find the destination going through all possible points.

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