Shourya Gupta
19095089
CSO-102
Electronics Engineering

1(a).
There are various other ways to use python dictionary in c but according to me using hashing is the best fit .

Hashing means using some function or algorithm to map object data to some representative integer value. This so-called hash code (or simply hash) can then be used as a way to narrow down our search when looking for the item in the map.
Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

eg:
in python→ dict = {'Name': 'Shourya', 'Age': 20, 'Role': 'devloper','language':'JavaScript'}

to save it in C the key will be converted to an integer key using a HashFunction and the corresponding value will be saved at that index using linkedList.(Direct Chaining teachnique)
To avoid collision hashFunction should be strong.

Hash Function
A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
A good hash function should have following properties
1) Efficiently computable.
2) Should uniformly distribute the keys (Each table position equally likely for each key)

Pseudo-Code
```
void hash(key,value){
 arr[128]  //create array of required size (either use malloac or give static size)
index=hash_function(key)   //→ will return an index less than 128
insert(index)          //→ will insert the value at that index one index can store more
than one value

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

1(b).

Index of a book is just an alphabetical sorted list of important words in the book along with their page numbers. Any useful index should not contain coomon or tiller words like articles, pronouns and other such things and we should filter out such words too.

Next we need to store the pages and since there can be multiple pages, an array would be required.

1. Checking for common word

   If we have a list of words to exclude then it would be only efficient if for each word we can decide if it is to be a part of index or not in O(1).

   for this we can use a hashmap in C++ .It would be used as an unordered_set <string>exclude .

   For each word in list

   exclude.insert(word)

   Now we can check in O(1) for presence in list by exclude.find(current word) .

2. A data structure that maintains sorted order internally and maps to an array of integers is required.

   Thus, any self balancing tree with nodes pointing to an array of integers would suffice. It allows logN insertion. In fact RB trees (Red-Black Tree)used by C++ internally in a map would be ideal for it.

C++ style pseude code would be

```
 unordered_set<string>exclude;
 for word in exclude_list;
    exclude.insert(word);
//Now O(1) queries for excluded check possible

 map<string, vector <int>>;
 for words in page:
   if ( exclude.find (word)===exclude.end())
   {
        index[wor].push_back(page_no);
    }
```

Complexity

1. Generally exclude hashmap

   O(m)*O(1)=O(m)

   where m is a list of words to be excluded

2. Building index

   O(n)*[O(1)+ O(logn)]

   n=no of words

   = O(nlogn)

   overalll complexity: O(m+nlogn)

   which would be theoritically best complexity as we would have to atleast traverse the exclude list and atleast sort the n words.

Printing index pseudo code

```
for (key in index)
{
 cout<<key<<" ";
 for (int page in index[key])
        cout<<page<<" ";
 cout<<"\n"
 }
```

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