CSCI 2270: Data Structures

Recitation #12 (Section 101)

Office Hours

- Name: Himanshu Gupta
  Email: himanshu.gupta@colorado.edu

- **Office Hours**
  - **12pm to 2pm on Mondays**
  - **12:30pm to 2:30pm on Fridays**
  - **Same Zoom ID as that of recitation - https://cuboulder.zoom.us/j/3112555724**

- In case that doesn't work for you, shoot me an email. We will figure something out that works for both of us.

Logistics

- In case you have a question during the recitation, just unmute yourself and speak up. Let’s try to make it as interactive as possible.

- You can ask questions via chat as well, but I would prefer if you guys ask your questions verbally.

- I would highly encourage you guys to switch on your cameras as well (if possible). It helps in making the session more lively and interactive.

Logistics

- **Midterm 2**
  - Scheduled on April 10 2020, 5 PM (Friday)

  - Tentative format:
    - Conceptual, short answer, multiple choice questions, 1 hour on Moodle from 5pm - 6pm. They are worth 30 points
    - Coding questions: due by midnight. They are worth 70 points.
    - Note - Heaps are not a part of this midterm.
    - Instructors and TA will be available to help till 8pm on zoom.

- **Final project**
  - It will be assigned next week, and will be due on April 26 2020.

Logistics

Make Up Assignment

- You have an opportunity to redo one of the assignments in which you have scored less points and get 100% in that assignment.
- You can choose any assignment from Assignment 5 to Assignment 9.
- Make sure the assignment runs perfectly fine on your local system.
- In Interview Grading, I will ask basic questions to check that you understood what you have done and to ensure you have done that yourself.
  - When? - Office Hours. Send me an email to reserve a time slot for 10-15 mins during my office hours.
  - Needs to be done during Week 15 (April 20th to April 26th)

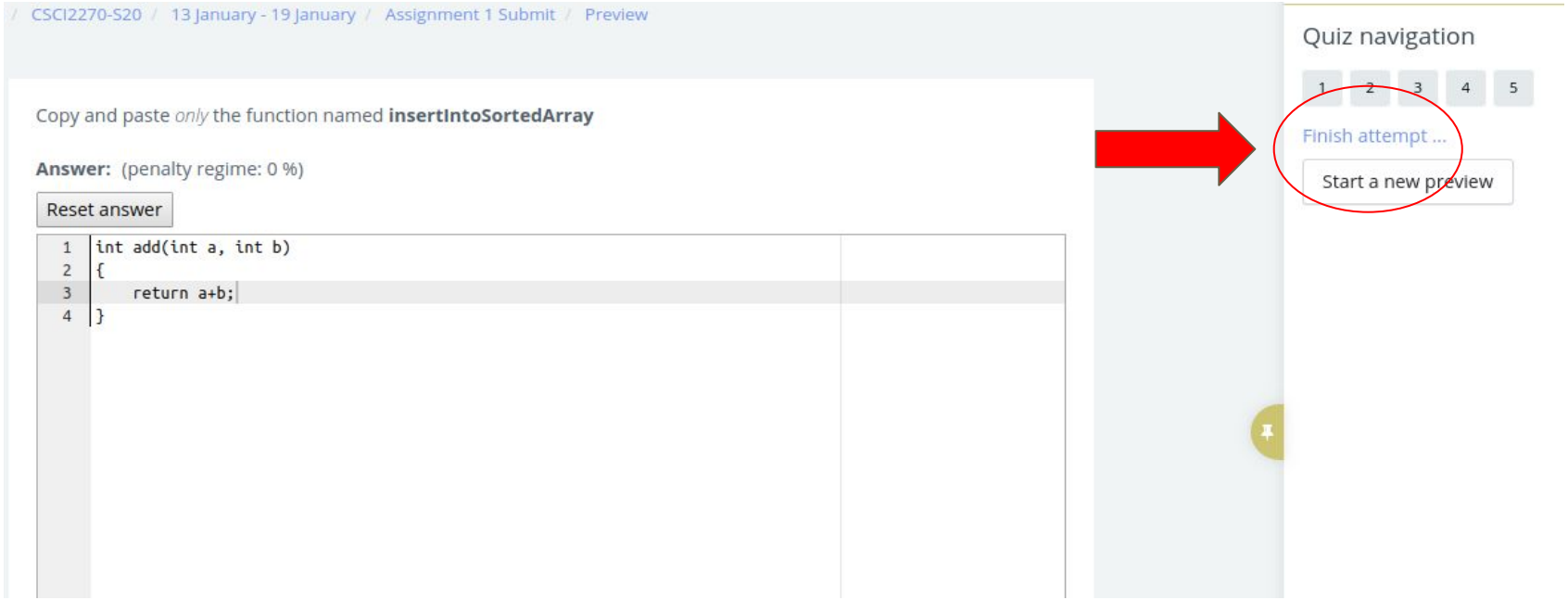
Logistics

- **Assignment 9 is due on Wednesday, April 15 2020, 11:59 PM**

  GOOD LUCK!

- **No recitation exercise this week! :D**

Please click on “Finish Attempt” after you are done!



Any questions?

Agenda

- Hashing
  - Motivation
  - Components of Hashing
- Reviewing Dijkstra’s algorithm
  - Algorithm
  - Intuition behind why it works
  - Implementation
- Review Practice Midterm 2
- Few practice questions
- Q&A session!

Hashing

- Why do we need hashing?

Hashing

- Why do we need hashing?
  - The write up has an excellent example of this.
  - Given a string S. How do you find the first repeated character in S?

Hashing

- Why do we need hashing?
  - The write up has an excellent example of this.
  - Given a string S. How do you find the first repeated character in S?

- Naive solution - For each character in the string check if the character is repeated or not.

Psuedocode -

```
void find_first_repeating_character(string S)
        for i in (0, length(S))
                for j in (0,length(S))
                        if( S[i] == S[j])
                                cout<<"First repeating character is " << S[i]<<endl;
                                return;
```

Hashing

- Why do we need hashing?
  - The write up has an excellent example of this.
  - Given a string S. How do you find the first repeated character in S?

- Naive solution - For each character in the string check if the character is repeated or not. **Time complexity - $O(n^2)$ where n is the length of S**

Psuedocode -

```
void find_first_repeating_character(string S)
        for i in (0, length(S))
                for j in (0,length(S))
                        if( S[i] == S[j])
                                cout<<"First repeating character is " << S[i]<<endl;
                                return;
```

Hashing

- Why do we need hashing?
  - The write up has an excellent example of this.
  - Given a string S. How do you find the first repeated character in S?

- Can we do better? If yes, how?

Hashing

- Why do we need hashing?
  - The write up has an excellent example of this.
  - Given a string S. How do you find the first repeated character in S?

- Can we do better? If yes, how?
  - Another naive solution -
    - We know there are only 256 possible ASCII characters. So, we can create a frequency array of size 256 which keeps track of the number of times we have seen that character. We initialize all the values in the frequency array to 0.
    - We then iterate through our string S character by character. We look at the ASCII value of the character (call it asc_value) and modify the value at the index asc_value in our frequency array from 0 to 1.
    - While doing this, if we encounter that the value for a character in the frequency array is already 1, then we know that we have seen this character earlier, and this is our answer. **This approach takes O(n) time.**

Hashing

- Why do we need hashing?
  - Now, let's modify the problem statement a bit.
  - Given an array of integers, find the first non repeating integer in that array.

- Can we use the same algorithm as last time?

Hashing

- Why do we need hashing?
  - Now, let's modify the problem statement a bit.
  - Given an array of integers, find the first non repeating integer in that array.

- Can we use the same algorithm as last time?
  - Well, NO.
  - Why not?
    - Earlier we knew that the number of distinct characters can't be more than 256, and so we created a frequency array of that size. However, in this case, the size of the array that we create should be greater than or equal to the maximum number in the given array.
    - This can lead to huge wastage of space and is not optimal.
      - For example - If the given array is [1, 100000000, 3, 5 , 1, 7]
      - We create a frequency array of size 100000000. However, that is not really needed.
      - This takes O(n) time and O(100000000) space.
      - **We are smart people. We can do better than this!**

Hashing

- Why do we need hashing?
  - Now, let's modify the problem statement a bit.
  - Given an array of integers, find the first non repeating integer in that array.

- Can we do better? If yes, how?
  - **Well, as it turns out, we can do this in O(n) time and less space, and we use the technique of hashing for doing that. We will look at the algorithm shortly.**
  - **We observed that addition, deletion or search in a balanced BST takes O(logn) time. With hashing, we can perform these operations in O(1). Worst case complexity of hashing is still O(n), but it is O(1) on average.**

Any questions?

Hashing

- What is hashing?
  - Hashing is a technique used for storing, retrieving and deleting information as fast as possible, fastest being O(1). Hashing can be used to uniquely identify a specific object from a group of similar objects.

  - Some examples of how hashing is used in our lives include:
    - In universities, how do you look for a particular student's details? Each student is assigned a unique roll number that can be used to retrieve information about him/her.
    - In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the number of copies we have in the library.
    - In these examples students and books are mapped/hashed to a unique number which we use to access information about that item.

Hashing

- So, getting back to the earlier problem of finding the first repeating integer in an array.

- Our problem was the huge space complexity of the frequency array because of large integer elements in the given array. So, we need a method to map these large elements/keys in the array to much smaller numbers.

- In hashing, large elements/keys are mapped/converted into small keys by using hash functions and the elements are then stored in a data structure called the hash table.

Hashing

- Example
  - Given array, A - [1, 100000000, 33, 25 , 1, 7]
    Suppose our hash function is this
    hash_func(int ele)
    return ele%10 //divide the element by 10 and return the remainder

  - So, hashed/mapped keys of every element in A using hash_func are

Element	1	100000000	33	25	7
hashed key	1	0	3	5	7

  - **Our new array(also known as hash table) can be of size 10 and store the count of these elements at their new hashed_key index.**

So, how to solve the earlier problem?

- Given an array of integers A, find the first non repeating integer in that array.

- Psuedocode
  - //Define a hash function
    int hash_func(int ele)
        return ele%10
  - Create a frequency array of size 10. Call it hash_table // int hash_table[10];
  - Initialize all values in hash_table to be 0
  - for element in A:
        hashed_key = hash_func(element)
        if(hash_table[hashed_key] == 1) // if it is 1, we know that we have seen this element before
            cout<<"First non repeating element in the array is" << element <<endl;
            return;
        else
            hash_table[hashed_key] = 1

Useful applications of Hashing

- Access student records with just student_id
  Trees - O(logn) ; Hash - O(1)

- Add records for a new student
  Trees - O(logn); Hash - O(1)

- Delete records for an old student
  Trees - O(logn); Hash - O(1)

Any questions?

- Is this approach clear to everyone?
- This is the most basic example of hashing.
- We will now discuss the various different components that exist in hashing (We already used some of them in the above example).

Components of Hashing

Hashing has four key components:

- Hash Table
- Hash Functions
- Collisions
- Collision Resolution Techniques

Let’s discuss them one by one very briefly. The write up has detailed explanation for all of them.

Hash Table

- In the earlier example, we observed how we can solve it by creating a large frequency array. However, it is not feasible to create such large arrays all the time. It is undesirable too because it is not optimal. So, we use hash tables instead.

- A hash table is a data structure that stores elements/records in a linear structure, such as an array or a vector. The index at which every element/record will be stored is calculated for each element/record by a hashing function.

Hash Function

- A hash function is a function that is used to transform/map the elements of an array into an index. Ideally, the hash function should map each possible element to a unique slot index, but it is difficult to achieve this in practice.

Collisions

- As mentioned in the previous slide, hash functions are used to map each element in an array to an index. Any operation associated with that element is performed at the hashed index in our hash table.

- However, it is observed that with any hash function, often more than one elements are mapped to the same index. This can cause problems.
    - For example, while adding records for a new student, if two student ids map to the same index, we don't know whose record to store there.

- Collision is this condition where two records/elements are mapped to the same index. Practically, it is not possible to create hash functions that don't have collisions. So, we should find some other way to fix this issue.

Collision Resolution Techniques

- How do we handle collisions in real life?
  - We let collisions happen and then resolve them.

- The process of finding an alternate location when a collision has occurred is called collision resolution.

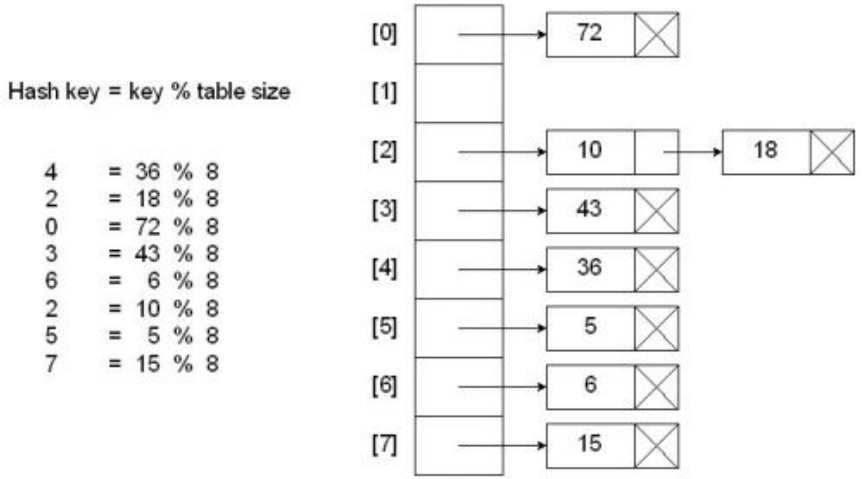
- Even though hash tables have collision problems, they are more efficient in many cases to all other data structures like search trees.

- There are many collision resolution techniques. For the purpose of this class we will study these two collision resolution techniques:
  - Direct Chaining
  - Open Addressing

Collision Resolution Techniques - Direct Chaining

- When two or more records/elements are hashed to the same index, these records are constituted into a singly-linked list called a chain at that index.

- For example - in the figure we are trying to insert few integers into a hash table using the given hash function, and we can see how collision is resolved when two or more integers are hashed to the same index.



Collision Resolution Techniques - Open Addressing

- In direct Chaining, the elements were stored in a new linked list at that index. However, in open addressing all keys are stored in the hash table itself. We resolve collisions by probing.

- We will specifically look at linear probing

Collision Resolution Techniques - Linear Probing

- We first calculate the hashed key/index of an element.
- If we see that a record already exists at that hashed index in the hash table, that means there is a collision.
- In linear probing, we search the hash table sequentially starting from the original hashed key/index. If a location is occupied, we check the next immediate location in the hash table. We keep doing this until we find an empty location.
- The code might look something like this -

```
index = hash(record.key)
while(hash_table[index] is occupied)
        index = (index + 1) % hash_table_size   // rehash function in the
        writeup
hash_table[index] = record
```

Collision Resolution Techniques - Linear Probing

- Problems with Linear Probing
    - One of the problems with linear probing is that table items tend to cluster together in the hash table.
    - This means that table contains groups of consequently occupied locations that are called clustering.
    - Clusters can get close to one another, and merge into a larger cluster. Thus, the one part of the table might be quite dense, even though another part has relatively few items.
    - Clustering causes long probe searches and therefore decreases the overall efficiency from O(1) to O(n).

Characteristics of a good hash function

- A good hash function should
  - minimise collision.
  - be easy and quick to compute.
  - distribute key values evenly in the hash table.
  - use all the information provided in the key

Any questions?

Dijkstra's Algorithm

- Last time,
  - We saw that BFS finds the shortest path between any two nodes in a given unweighted graph. However, BFS fails to find the smallest cost path on a weighted graph.
  - Since most of the graphs in real life problems are weighted, we need an algorithm that can find the smallest cost path between any two nodes in a weighted graph.
  - **Dijkstra's algorithm is one such algorithm.**
  - Other commonly used algorithms are - Floyd–Warshall's Algorithm and Bellman Ford's Algorithm. They are beyond the scope of this course. You will learn them if you take CSCI 3104 - Algorithms.
  - If you want to talk about them, feel free to discuss it with me personally.

Dijkstra's Algorithm

- There are multiple different implementations of it. The most common one is using priority queue. However, we will discuss the implementation discussed in class by the instructor.

- I have modified the algorithm a bit to keep track of the predecessor node as well. **You need exactly this for Assignment 9!**

Dijkstra's Algorithm

- **Psuedocode**

```
Let the start vertex be src. Modify its visited flag to be True
Create a list called solved_list. // This list keeps track of all the nodes and their distance from the start vertex and their predecessor node.
Insert src, 0, NULL to the solved_list
while( not all nodes have been visited)
        Create a new list called temporary_list
        for x in solved_list // traverse the entire solved_list
                find the adjacent unvisited node of x that is the closest to x. Let's call this node
                closest_to_x  // Note that predecessor of closest_to_x is is x
                Calculate the distance for closest_to_x .Call it dist
                //dist = x.distance + weight of the edge (x,closest_to_x)
                Add closest_to_x node with its distance(dist) and its predecessor(x) in the temporary_list
                // closest_to_x, dist, x added to temporary_list.
        Loop through all the entries in temporary_list and find that node which  has the minimum dist value. Let's call this node to_be_added
        Modify distance, visited flag and pred of the node to_be_added
        //  to_be_added. visited = True; to_be_added.distance = corresponding dist value from temporary_list
        //  to_be_added.pred = corresponding x value from temporary_list
        Add to_be_added, to_be_added.distance, to_be_added.pred to solved_list
```

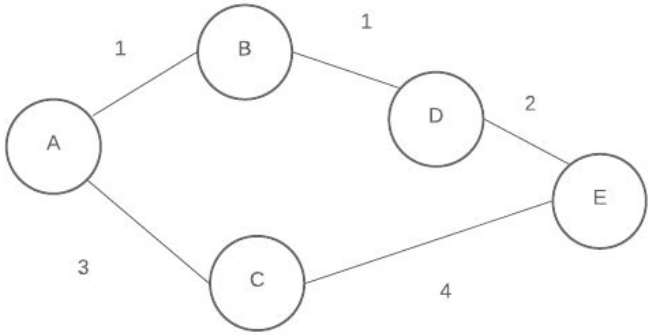
Let’s apply this algorithm to the same weighted graph we saw in last recitation and see the results

Dijkstra's Algorithm to find length of shortest path on a weighted graph

Initialisation of different variables

Start vertex or src = A

solved_list



Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	False	False	False	False	False
Pred	NULL	NULL	NULL	NULL	NULL

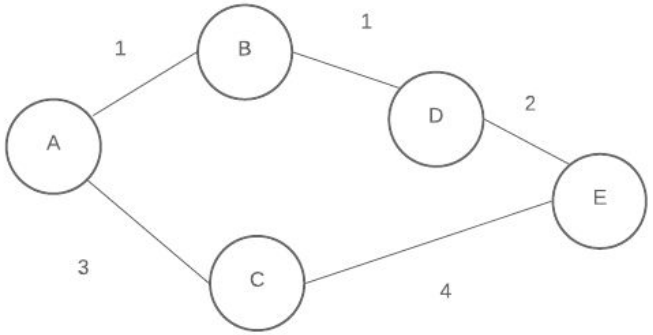
Dijkstra's Algorithm to find length of shortest path on a weighted graph

Initialisation of different variables

Start vertex or src = A

solved_list

(A,0,NULL)

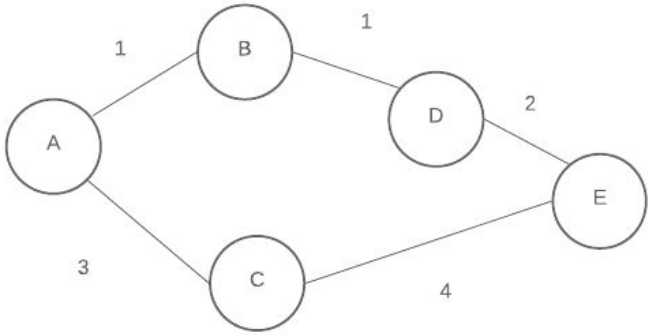


Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	True	False	False	False	False
Pred	NULL	NULL	NULL	NULL	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

Initialisation of different variables

Start vertex or src = A



solved_list

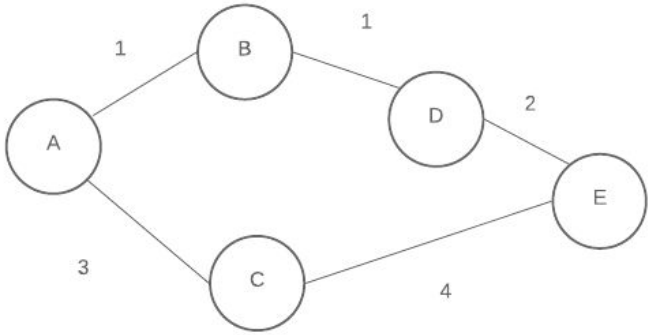
(A,0,NULL)

Traverse through all entries in solved_list and generate your temporary_list

Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	True	False	False	False	False
Pred	NULL	NULL	NULL	NULL	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

(A,0,NULL)

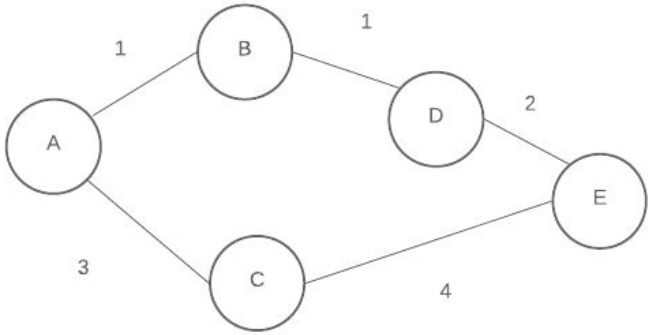
temporary_list

(B,1,A)

Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	True	False	False	False	False
Pred	NULL	NULL	NULL	NULL	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

(A,0,NULL)

temporary_list

(B,1,A)

Find the node with the minimum dist value in temporary_list and insert that node in your solved_list

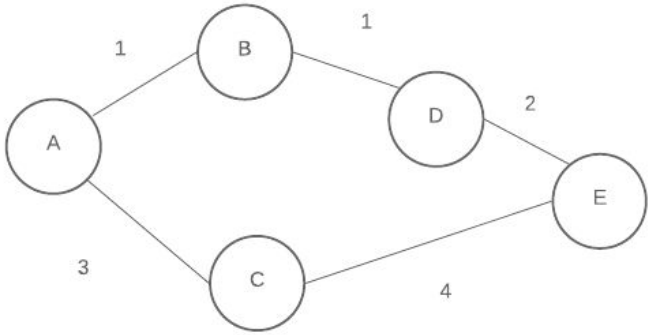
Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	True	False	False	False	False
Pred	NULL	NULL	NULL	NULL	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A

solved_list

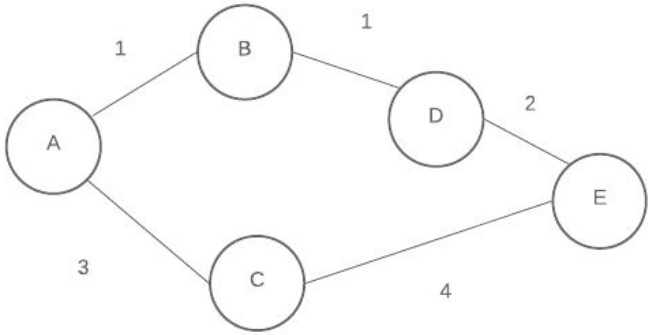
(A,0,NULL)	(B,1,A)



Vertex	A	B	C	D	E
Distance	0	1	0	0	0
Visited	True	True	False	False	False
Pred	NULL	A	NULL	NULL	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

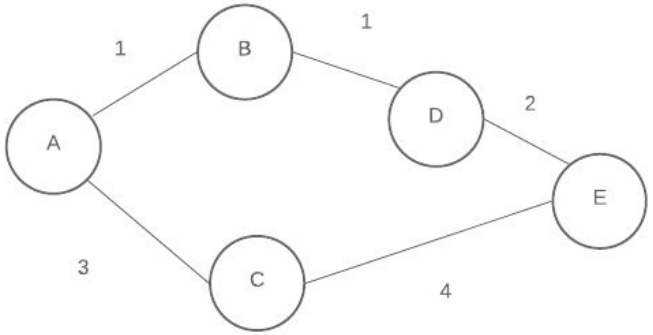
(A,0,NULL)	(B,1,A)

Traverse through all entries in solved_list and generate your temporary_list

Vertex	A	B	C	D	E
Distance	0	1	0	0	0
Visited	True	True	False	False	False
Pred	NULL	A	NULL	NULL	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

(A,0,NULL)	(B,1,A)

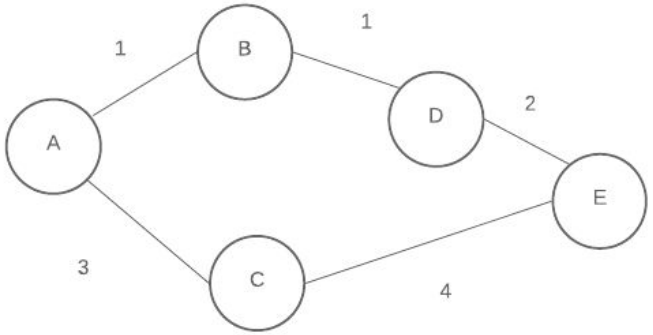
temporary_list

(C,3,A)	(D,2,B)

Vertex	A	B	C	D	E
Distance	0	1	0	0	0
Visited	True	True	False	False	False
Pred	NULL	A	NULL	NULL	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

(A,0,NULL)	(B,1,A)

temporary_list

(C,3,A)	(D,2,B)

Find the node with the minimum dist value in temporary_list and insert that node in your solved_list

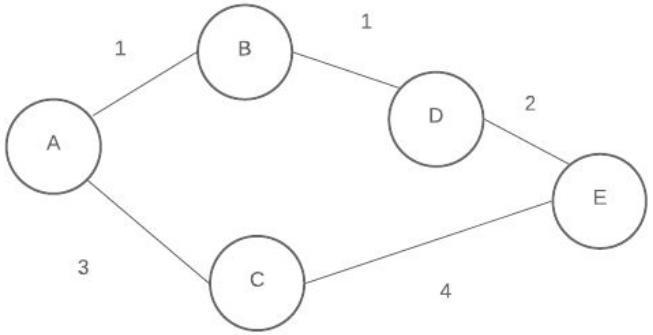
Vertex	A	B	C	D	E
Distance	0	1	0	0	0
Visited	True	True	False	False	False
Pred	NULL	A	NULL	NULL	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A

solved_list

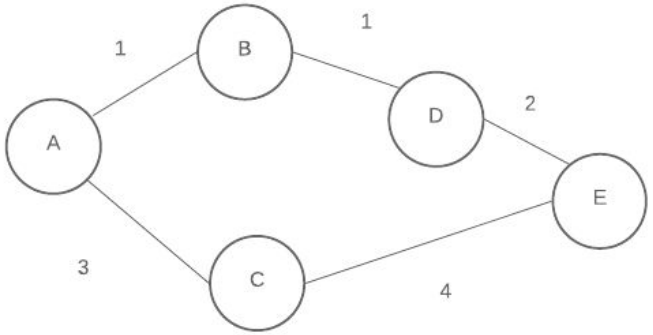
(A,0,NULL)	(B,1,A)	(D,2,B)



Vertex	A	B	C	D	E
Distance	0	1	0	2	0
Visited	True	True	False	True	False
Pred	NULL	A	NULL	B	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

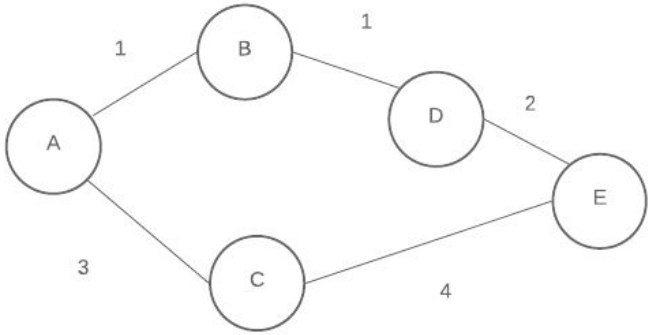
(A,0,NULL)	(B,1,A)	(D,2,B)

Traverse through all entries in solved_list and generate your temporary_list

Vertex	A	B	C	D	E
Distance	0	1	0	2	0
Visited	True	True	False	True	False
Pred	NULL	A	NULL	B	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

(A,0,NULL)	(B,1,A)	(D,2,B)

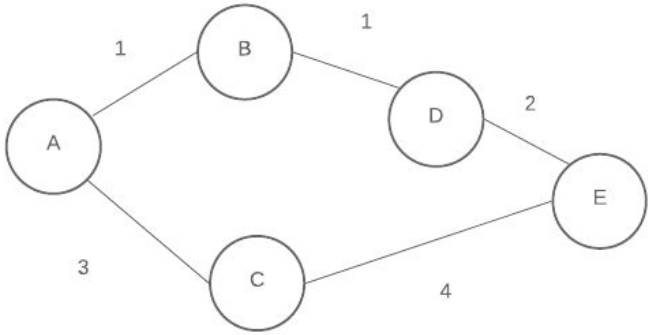
temporary_list

(C,3,A)	(E,4,D)

Vertex	A	B	C	D	E
Distance	0	1	0	2	0
Visited	True	True	False	True	False
Pred	NULL	A	NULL	B	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

(A,0,NULL)	(B,1,A)	(D,2,B)

temporary_list

(C,3,A)	(E,4,D)

Find the node with the minimum dist value in temporary_list and insert that node in your solved_list

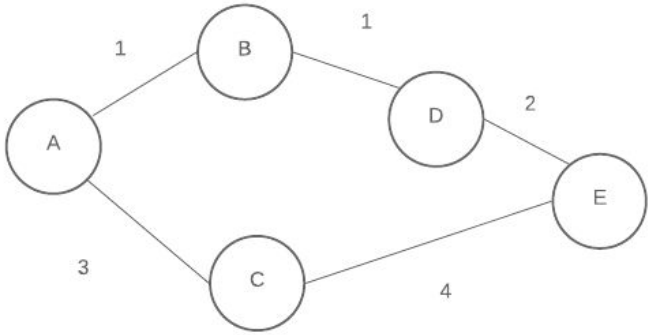
Vertex	A	B	C	D	E
Distance	0	1	0	2	0
Visited	True	True	False	True	False
Pred	NULL	A	NULL	B	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A

solved_list

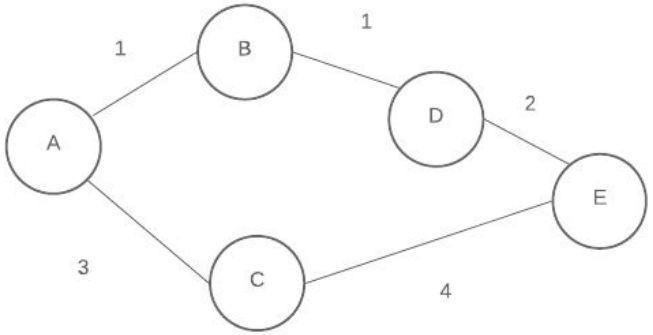
(A,0,NULL)	(B,1,A)	(D,2,B)	(C,3,A)



Vertex	A	B	C	D	E
Distance	0	1	3	2	0
Visited	True	True	True	True	False
Pred	NULL	A	A	B	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

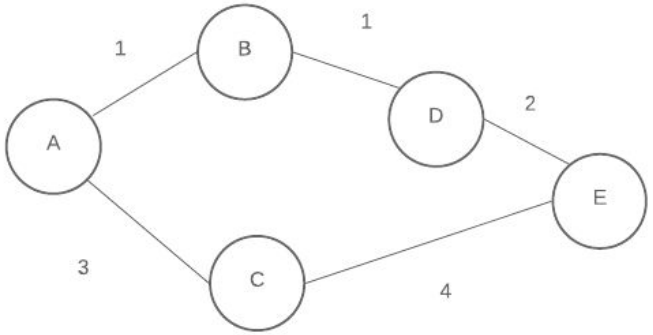
(A,0,NULL)	(B,1,A)	(D,2,B)	(C,3,A)

Traverse through all entries in solved_list and generate your temporary_list

Vertex	A	B	C	D	E
Distance	0	1	3	2	0
Visited	True	True	True	True	False
Pred	NULL	A	A	B	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

(A,0,NULL)	(B,1,A)	(D,2,B)	(C,3,A)

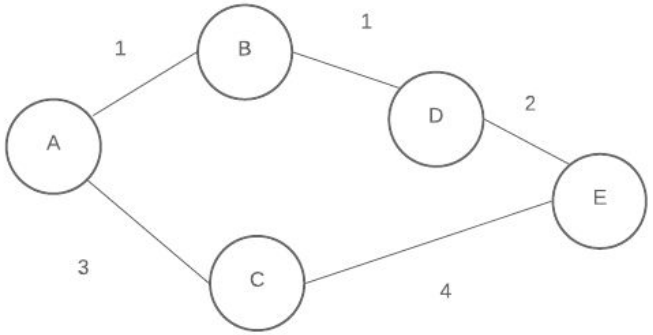
temporary_list

(E,7,C)	(E,4,D)

Vertex	A	B	C	D	E
Distance	0	1	3	2	0
Visited	True	True	True	True	False
Pred	NULL	A	A	B	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

(A,0,NULL)	(B,1,A)	(D,2,B)	(C,3,A)

temporary_list

(E,7,C)	(E,4,D)

Find the node with the minimum dist value in temporary_list and insert that node in your solved_list

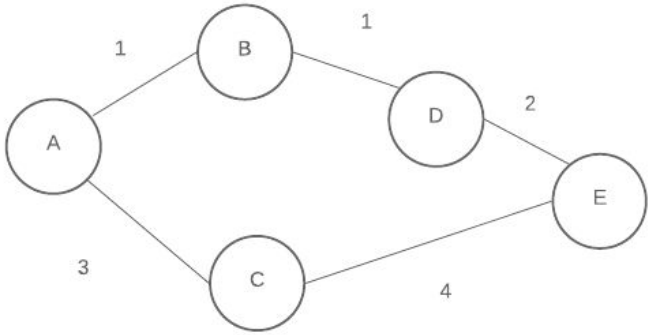
Vertex	A	B	C	D	E
Distance	0	1	3	2	0
Visited	True	True	True	True	False
Pred	NULL	A	A	B	NULL

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A

solved_list

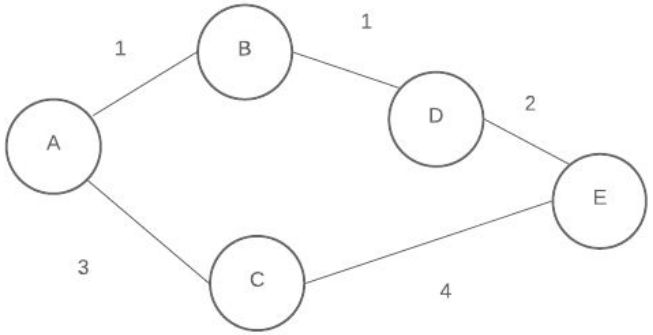
(A,0,NULL)	(B,1,A)	(D,2,B)	(C,3,A)	(E,4,D)



Vertex	A	B	C	D	E
Distance	0	1	3	2	4
Visited	True	True	True	True	True
Pred	NULL	A	A	B	D

Dijkstra's Algorithm to find length of shortest path on a weighted graph

start vertex or src = A



solved_list

(A,0,NULL)	(B,1,A)	(D,2,B)	(C,3,A)	(E,4,D)

No more unvisited vertices and so the algorithm stops!

Vertex	A	B	C	D	E
Distance	0	1	3	2	4
Visited	True	True	True	True	True
Pred	NULL	A	A	B	D

Any questions about this implementation?

Intuition behind why Dijkstra's algorithm works

- We have the following steps in our psuedocode

for x in solved_list // traverse the entire solved_list

find the adjacent unvisited node of x that is the closest to x. Let's call this node *closest_to_x* // Note that predecessor of closest_to_x is is x

Calculate the distance for *closest_to_x* .Call it dist

//dist = x.distance + weight of the edge (x,closest_to_x)

Add *closest_to_x* node with its distance(dist) and its predecessor(x) in the temporary_list

// closest_to_x, dist, x added to temporary_list.

Loop through all the entries in temporary_list and find that node which has the minimum dist value. Let's call this node *to_be_added*

- Can someone tell me what does this set of steps do?

Intuition behind why Dijkstra's algorithm works

- We have the following steps in our psuedocode

for x in solved_list // traverse the entire solved_list

find the adjacent unvisited node of x that is the closest to x. Let's call this node *closest_to_x* // Note that predecessor of closest_to_x is is x

Calculate the distance for *closest_to_x* .Call it dist

//dist = x.distance + weight of the edge (x,closest_to_x)

Add *closest_to_x* node with its distance(dist) and its predecessor(x) in the temporary_list

// closest_to_x, dist, x added to temporary_list.

Loop through all the entries in temporary_list and find that node which has the minimum dist value. Let's call this node *to_be_added*

- Can someone tell me what does this set of steps do?
  - For every node in the list of nodes that have been solved, it looks at all the possible adjacent nodes that haven't been visited yet and adds the closest one to temporary_list.
  - So, temporary_list is a collection of all new nodes that can be visited from existing solved nodes.
  - We pick the node with the smallest distance from this list and add it to the solved_list

Intuition behind why Dijkstra's algorithm works

- It's a tricky concept. Its okay if you didn't get it.
- This will be covered again in the Algorithms course with a proper mathematical proof and that will make it much more clearer.

Practice Midterm

Practice Midterm

Problem 1 - Given a Binary Search Tree, return the sum of values of all the nodes that fall within a range [min, max] inclusive.

What should be your approach?

Practice Midterm

Problem 1 - Given a Binary Search Tree, return the sum of values of all the nodes that fall within a range [min, max] inclusive.

Write your own helper function to do recursive calls!

```
int sum_within_range(node *node, int min, int max){
        if(node==NULL)
                return 0;
        if (node->key >= min && node->key <=max)
                return node->key+ sum_within_range(node->left,min,max) +
                                        sum_within_range(node->right,min,max);
        else
                return sum_within_range(node->left,min,max) +
                                sum_within_range(node->right,min,max);
```

Practice Midterm

Problem 3 - Given an unweighted graph and a starting vertex, count the number of nodes a specified distance away.
Given a starting node with value key and a distance dist , return the number of nodes that have a path from id with exactly dist edges. If there are multiple paths, only consider the shortest one.

```
int countNodesWithDist(int id, int dist); // example declaration
```

What should be your approach?

Practice Midterm

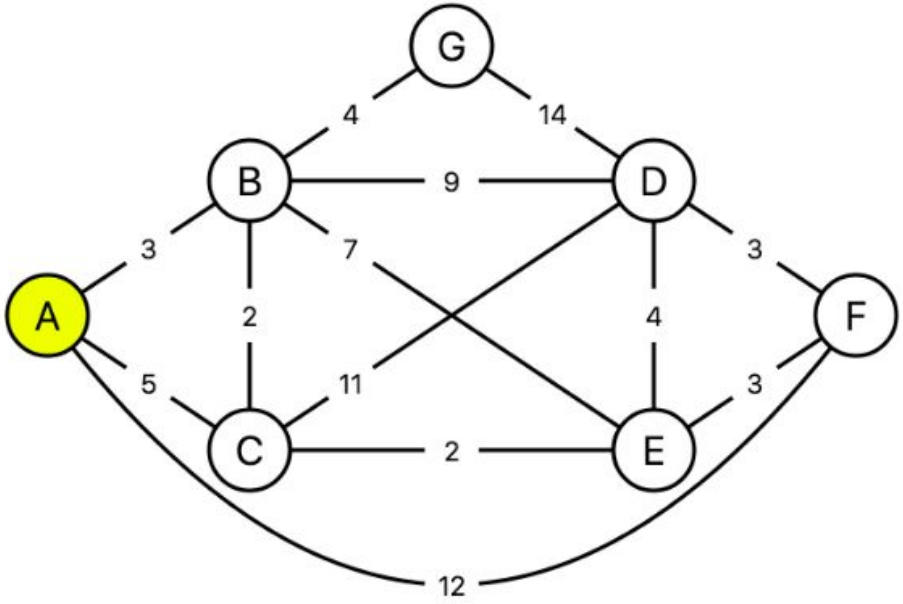
Problem 3 - Given an unweighted graph and a starting vertex, count the number of nodes a specified distance away.
Given a starting node with value key and a distance dist , return the number of nodes that have a path from id with exactly dist edges. If there are multiple paths, only consider the shortest one.

```
int countNodesWithDist(int id, int dist); // example declaration
```

What should be your approach?
Since the given graph is unweighted, just run BFS algorithm from start node to find length of shortest path to all the nodes.
After this, just return the number of nodes that have their distance = dist

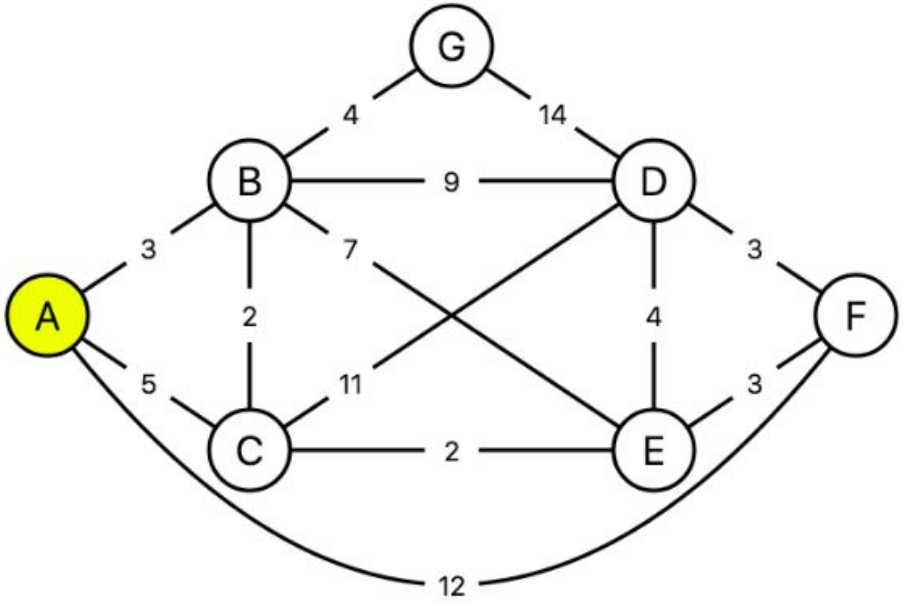
Practice Questions

Q. What is the shortest path from A to F?



Practice Questions

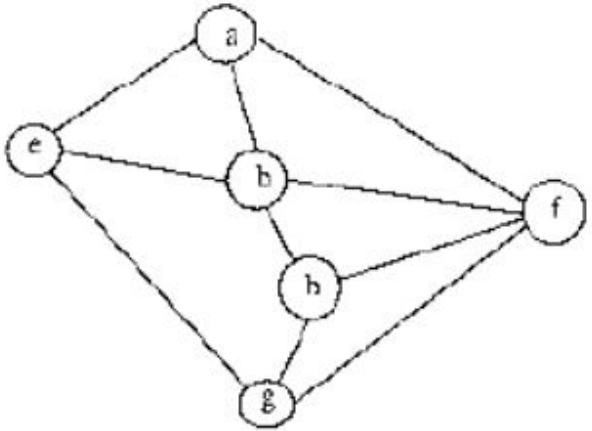
Q1. What is the shortest path from A to F?



Answer: A->C->E->F

Practice Questions

Q. Consider the graph on the right.



Among the following sequences:

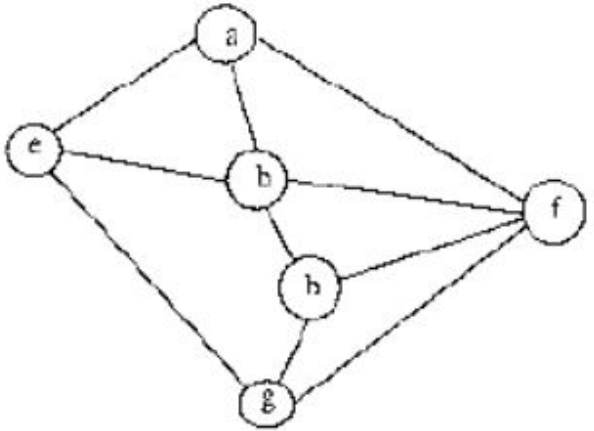
(I) a b e g h f
(II) a b f e h g
(III) a b f h g e
(IV) a f g h b e

Which are valid depth first traversals of the above graph?

A) I, II and IV only
B) I and IV only
C) II, III and IV only
D) I, III and IV only

Practice Questions

Q. Consider the graph on the right.



Among the following sequences:

(I) a b e g h f
(II) a b f e h g
(III) a b f h g e
(IV) a f g h b e

Which are valid depth first traversals of the above graph?

A) I, II and IV only
B) I and IV only
C) II, III and IV only
D) I, III and IV only

ANSWER: D)

That's it!

- Well, that's it from my side today.
- You can expect MCQs like these in your midterm. So, do practice them.
- Q&A session -
  I will stay here on the zoom call till 9:15 am for any questions that you might have. Feel free to ask any question that might be bothering you.

GOOD LUCK FOR THE MIDTERM