01. Write a Prolog program to merge two ordered list generating an ordered list.

Code:

```
% Base cases:
merge([], L, L). % If the first list is empty, the merged list is the second list.
merge(L, [], L). % If the second list is empty, the merged list is the first list.

% Recursive case:
merge([H1|T1], [H2|T2], [H1|T]) :-
    H1 =< H2, % If the head of the first list is less than or equal to the head of the second list,
    merge(T1, [H2|T2], T). % Recursively merge the rest of the lists.

merge([H1|T1], [H2|T2], [H2|T]) :-
    H1 > H2, % If the head of the first list is greater than the head of the second list,
    merge([H1|T1], T2, T). % Recursively merge the rest of the lists.
```

Explain:

Base cases:

1. merge([], L, L).
    - This rule says that if the first list is empty ([]), the merged result is simply the second list (L). There's nothing left to merge.
2. merge(L, [], L).
    - This rule says that if the second list is empty ([]), the merged result is the first list (L). Again, nothing left to merge.

Recursive cases:

1. merge([H1|T1], [H2|T2], [H1|T]) :- H1 =< H2, merge(T1, [H2|T2], T).
    - If the head of the first list (H1) is smaller than or equal to the head of the second list (H2), include H1 in the merged result. Then, recursively merge the tail of the first list (T1) with the second list ([H2|T2]).
2. merge([H1|T1], [H2|T2], [H2|T]) :- H1 > H2, merge([H1|T1], T2, T).
    - If the head of the first list (H1) is greater than the head of the second list (H2), include H2 in the merged result. Then, recursively merge the first list ([H1|T1]) with the tail of the second list (T2).

Base Cases:

1. **merge([], L, L)**:
    - **Explanation**: This rule handles the scenario where the first list is empty. If the first list is [], the merged result is simply the second list (L).
    - **Why?**: Since there is nothing left in the first list to merge, the result is whatever is left in the second list.
2. **merge(L, [], L)**:
    - **Explanation**: This rule handles the scenario where the second list is empty. If the second list is [], the merged result is simply the first list (L).
    - **Why?**: Since there is nothing left in the second list to merge, the result is whatever is left in the first list.

Recursive Cases:

The recursive cases handle merging when neither list is empty. The logic is driven by comparing the heads of both lists (H1 from the first list, and H2 from the second list).

1. **merge([H1|T1], [H2|T2], [H1|T]) :- H1 =< H2, merge(T1, [H2|T2], T)**:
    - **Explanation**:
        - If the head of the first list (H1) is less than or equal to the head of the second list (H2), then H1 is included in the merged result.
        - After including H1, the program recursively calls merge to continue merging the tail of the first list (T1) with the entire second list ([H2|T2]).
    - **Why?**: Since H1 is smaller or equal to H2, it should come first in the merged result, and we move forward with the remaining elements.
2. **merge([H1|T1], [H2|T2], [H2|T]) :- H1 > H2, merge([H1|T1], T2, T)**:
    - **Explanation**:
        - If the head of the first list (H1) is greater than the head of the second list (H2), then H2 is included in the merged result.
        - After including H2, the program recursively calls merge to continue merging the first list ([H1|T1]) with the tail of the second list (T2).
    - **Why?**: Since H1 is greater than H2, H2 should come first in the merged result, and we continue with the remaining elements.

Detailed Recursive Working Example

Consider merging the two ordered lists: [1, 3, 5] and [2, 4, 6].

Step-by-Step Process:

1. **First call:**

    prolog
    Copy code
    merge([1, 3, 5], [2, 4, 6], MergedList).

    - Heads are H1 = 1 and H2 = 2.
    - Since H1 = 1 <= H2 = 2, the merged result starts with 1, and the function recursively merges the remaining lists:

        merge([3, 5], [2, 4, 6], T).
        MergedList = [1 | T].

2. **Second call:**

    merge([3, 5], [2, 4, 6], T).

    - Now H1 = 3 and H2 = 2.
    - Since H1 = 3 > H2 = 2, the merged result starts with 2, and the function recursively merges the remaining lists:

        merge([3, 5], [4, 6], T2).

T = [2 | T2].

3. **Third call:**

```
prolog
Copy code
merge([3, 5], [4, 6], T2).
```

- Now H1 = 3 and H2 = 4.
- Since H1 = 3 <= H2 = 4, the merged result starts with 3, and the function recursively merges the remaining lists:

```
prolog
Copy code
merge([5], [4, 6], T3).
T2 = [3 | T3].
```

4. **Fourth call:**

```
prolog
Copy code
merge([5], [4, 6], T3).
```

- Now H1 = 5 and H2 = 4.
- Since H1 = 5 > H2 = 4, the merged result starts with 4, and the function recursively merges the remaining lists:

```
prolog
Copy code
merge([5], [6], T4).
T3 = [4 | T4].
```

5. **Fifth call:**

```
prolog
Copy code
merge([5], [6], T4).
```

- Now H1 = 5 and H2 = 6.
- Since H1 = 5 <= H2 = 6, the merged result starts with 5, and the function recursively merges the remaining lists:

```
prolog
Copy code
merge([], [6], T5).
T4 = [5 | T5].
```

6. **Sixth call (base case):**

```
merge([], [6], T5).
```

- Since the first list is now empty ([]), the result is simply the second list [6].
- Therefore, T5 = [6].

Returning from Recursion:

Now that we have reached the base case, the recursive calls start returning, constructing the final merged list step by step:

- Fifth call: T4 = [5, 6].
- Fourth call: T3 = [4, 5, 6].
- Third call: T2 = [3, 4, 5, 6].
- Second call: T = [2, 3, 4, 5, 6].
- First call: MergedList = [1, 2, 3, 4, 5, 6].

02. Write a Prolog program to concatenate two lists giving third list.

```
% Base case:
concatenate([], L, L). % If the first list is empty, the concatenated list is just the second list.

% Recursive case:
concatenate([H|T], L, [H|R]) :-
    concatenate(T, L, R). % Concatenate the tail of the first list with the second list.
```

Explanation:

- **Base case:** concatenate([], L, L).
    - If the first list is empty ([]), the result is simply the second list (L). This means that when we reach the end of the first list, the rest of the second list is added to the result.
- **Recursive case:** concatenate([H|T], L, [H|R]) :- concatenate(T, L, R).
    - The head of the first list (H) is added to the result list ([H|R]).
    - Then, the program recursively concatenates the tail of the first list (T) with the second list (L) and stores the result in R.
    -

Recursive Case:
The main working principle for the recursive case in the program is as follows:

```
concatenate([H|T], L, [H|R]) :-
    concatenate(T, L, R).
```

- The first argument is a list represented as [H|T], where H is the head (first element), and T is the tail (the rest of the list).
- The second argument (L) is the list to be concatenated to the first one.
- The third argument ([H|R]) is the resulting concatenated list, which starts with the head (H) of the first list, followed by the recursive concatenation of the tail (T) and the second list (L).

How Recursion Works:
Recursion breaks down the first list element by element, then adds each element to the result list until the first list is exhausted (empty). Let's walk through an example:

Example: Concatenating [1, 2, 3] with [4, 5, 6].

1. **First Call:**

   ```
   concatenate([1, 2, 3], [4, 5, 6], Result).
   ```

   This matches the recursive rule. H = 1, T = [2, 3], L = [4, 5, 6], so it proceeds to:

   ```
   concatenate([2, 3], [4, 5, 6], R).
   Result = [1 | R].
   ```

   The result will be [1 | R], where R will be determined by the next recursive call.

2. **Second Call:**

   ```
   concatenate([2, 3], [4, 5, 6], R).
   ```

   Now, H = 2, T = [3], so:

   ```
   concatenate([3], [4, 5, 6], R2).
   R = [2 | R2].
   ```

   The result will be [2 | R2], where R2 is determined by the next recursive call.

3. **Third Call:**

   ```
   concatenate([3], [4, 5, 6], R2).
   ```

   Here, H = 3, T = [], so:

   ```
   concatenate([], [4, 5, 6], R3).
   R2 = [3 | R3].
   ```

   The result will be [3 | R3], and now we reach the base case.

4. **Base Case:**

   ```
   concatenate([], [4, 5, 6], [4, 5, 6]).
   ```

   This matches the base case because the first list is empty. Therefore, the result is simply the second list:

   ```
   R3 = [4, 5, 6].
   ```

5. **Returning from Recursion:** Now that the base case has been reached, the recursive calls start returning:
   - From the third call: R2 = [3, 4, 5, 6]
   - From the second call: R = [2, 3, 4, 5, 6]
   - From the first call: Result = [1, 2, 3, 4, 5, 6]

Thus, the final concatenated list is [1, 2, 3, 4, 5, 6].

3. Write a Program that performs the following functions.
i. Concatenate two lists.
Ans: Same as 2 number questions
ii. Check an element is a member of a given list or not.

```
% Base case:
member(X, [X|_]).  % If the head of the list is the element, then it's a member.

% Recursive case:
member(X, [_|T]) :-
    member(X, T).  % Recursively check the tail of the list.
```

Recursive Case:

```
member(X, [_|T]) :-
   member(X, T).
```

- **What does this do?**: This is the recursive case. It says that if the element X is not the head of the list, then check if X is a member of the tail of the list (T).
- **How does this work?**:
  - The head of the list (represented by the underscore _) is ignored since it doesn't match X.
  - The recursive call checks the remaining part of the list (the tail, T).
  - The process continues until the base case is reached (either X matches the head of some tail, or the list becomes empty).

Detailed Recursive Working:
Let's go through a detailed step-by-step example:

Example Query:

```
?- member(4, [1, 2, 3, 4, 5]).
```

1. **First Call:**

   ```
   member(4, [1, 2, 3, 4, 5]).
   ```

   - Prolog compares the head of the list (1) with the target element (4).
   - Since 1 \= 4, the recursive case is triggered, and Prolog checks the tail:

     ```
     member(4, [2, 3, 4, 5]).
     ```

2. **Second Call:**

   ```
   member(4, [2, 3, 4, 5]).
   ```

   - Now, the head is 2, and Prolog compares it with 4.
   - Since 2 \= 4, the recursive case continues, and Prolog checks the tail:

     ```
     member(4, [3, 4, 5]).
     ```

3. **Third Call:**

   member(4, [3, 4, 5]).

   - The head is 3, which is still not equal to 4.
   - Prolog moves on and checks the tail:

     member(4, [4, 5]).

4. **Fourth Call (Base Case):**

   member(4, [4, 5]).

   - Now, the head of the list is 4, which matches the target element.
   - This triggers the base case, and Prolog returns true.

iii. Reverse a list.

```
% Base case:
reverse_list([], []). % An empty list is already reversed.

% Recursive case:
reverse_list([H|T], ReversedList) :-
    reverse_list(T, ReversedTail), % Recursively reverse the tail
    append(ReversedTail, [H], ReversedList). % Append the head at the end of the reversed tail.
```

Example:

Let's take an example of reversing a list [1, 2, 3, 4]:

?- reverse_list([1, 2, 3, 4], ReversedList).

1. **First Call:**

   reverse_list([1, 2, 3, 4], ReversedList).

   - The head is 1 and the tail is [2, 3, 4].
   - The program recursively calls reverse_list([2, 3, 4], ReversedTail).

2. **Second Call:**

   reverse_list([2, 3, 4], ReversedTail).

   - The head is 2 and the tail is [3, 4].
   - The program recursively calls reverse_list([3, 4], ReversedTail).

3. **Third Call:**

   reverse_list([3, 4], ReversedTail).

   - The head is 3 and the tail is [4].
   - The program recursively calls reverse_list([4], ReversedTail).

4. **Fourth Call:**

   reverse_list([4], ReversedTail).

   - The head is 4 and the tail is [].

- The program recursively calls reverse_list([], ReversedTail).

5. **Fifth Call (Base Case):**

   reverse_list([], []).

   - The base case is reached. The reversed empty list is still an empty list. This result is returned to the previous call.
6. **Returning to the Fourth Call:**
   - The reversed tail is now [], and the head 4 is appended to it.
   - This gives [4].
7. **Returning to the Third Call:**
   - The reversed tail is now [4], and the head 3 is appended to it.
   - This gives [4, 3].
8. **Returning to the Second Call:**
   - The reversed tail is now [4, 3], and the head 2 is appended to it.
   - This gives [4, 3, 2].
9. **Returning to the First Call:**
   - The reversed tail is now [4, 3, 2], and the head 1 is appended to it.
   - This gives [4, 3, 2, 1].

Thus, the list [1, 2, 3, 4] is reversed to [4, 3, 2, 1].

iv. Delete an element from a list

```
list_delete(X,[X],[]).
list_delete(X,[X|L1],L1).
list_delete(X,[Y|L2],[Y|L1]):- list_delete(X,L2,L1).
```

Recursive Workflow:
Let's go through an example where we want to delete the element 2 from the list [1, 2, 3, 4]:

Query:

```
?- list_delete(2, [1, 2, 3, 4], L).
```

1. **First Call:**

   list_delete(2, [1, 2, 3, 4], L).

   - The head is 1, and it does not match 2.
   - The rule list_delete(X, [Y|L2], [1|L1]) is applied.
   - The head 1 is retained in the resulting list, and the program recursively calls:

     list_delete(2, [2, 3, 4], L1).

2. **Second Call:**

   list_delete(2, [2, 3, 4], L1).

- The head is 2, which matches the element to be deleted.
- The rule list_delete(X, [X|L1], L1) is applied, removing 2 and returning the tail [3, 4] as the result.

3. **Returning to First Call:**
    - The result from the second call is [3, 4].
    - The head 1 is retained, so the final result is:

      L = [1, 3, 4].

4.Write a Prolog program to add an element in a head position in given list.

add_head(Element, List, [Element|List]).

Explanation:

- add_head/3 is a predicate that takes three arguments:
    1. Element: The element to be added at the head of the list.
    2. List: The original list to which the element will be added.
    3. [Element|List]: The resulting list with Element prepended to the original list.
- The | operator in Prolog is used to prepend an element (Element) to the front of an existing list (List).

Example Queries:

1. Add the element 5 to the head of the list [1, 2, 3]:

   ?- add_head(5, [1, 2, 3], Result).
   Result = [5, 1, 2, 3].

2. Add the element a to an empty list:

   ?- add_head(a, [], Result).
   Result = [a].

add_head(Element, List, [Element|List]).

5. Write a Prolog program to add an element in a last position in given list.

```
add_last(Element, [], [Element]). % Base case: If the list is empty, the result is a list with just the element.
 add_last(Element, [H|T], [H|NewT]) :- add_last(Element, T, NewT).  % Recursive case: Traverse through the list and keep the structure.
```

Explanation:

- add_last/3 is a predicate with three arguments:
    1. Element: The element to be added at the end of the list.
    2. [H|T]: The original list, where H is the head and T is the tail.
    3. [H|NewT]: The resulting list, which will have the same head (H), and the tail (NewT) is the result of adding the element to the rest of the list.

Base Case:

```
add_last(Element, [], [Element]).
```

- This rule is the base case for recursion. It says that if the list is empty ([]), the result is a new list containing only the element ([Element]).
- **Example**: If the list is empty, and we want to add 5, the result will be [5].

Recursive Case:

```
add_last(Element, [H|T], [H|NewT]) :-
   add_last(Element, T, NewT).
```

- This rule works by recursively traversing the list. It takes the head (H) of the list and keeps it the same, while the tail (T) is processed recursively.
- The recursion continues until the base case is reached (an empty list), and the element is finally added there.

Example Queries:

1. **Add 5 to the end of the list [1, 2, 3, 4]:**

   ```
   ?- add_last(5, [1, 2, 3, 4], Result).
   Result = [1, 2, 3, 4, 5].
   ```

2. **Add a to an empty list:**

   ```
   ?- add_last(a, [], Result).
   Result = [a].
   ```

Recursive Workflow:

Let’s see how the recursive flow works for adding the element 5 to the list [1, 2, 3].

Query:

?- add_last(5, [1, 2, 3], Result).

1. **First Call:**

   add_last(5, [1, 2, 3], Result).

   - The head is 1, and the tail is [2, 3].
   - The program recurses with add_last(5, [2, 3], NewT).
2. **Second Call:**

   add_last(5, [2, 3], NewT).

   - The head is 2, and the tail is [3].
   - The program recurses with add_last(5, [3], NewT).
3. **Third Call:**

   add_last(5, [3], NewT).

   - The head is 3, and the tail is an empty list ([]).
   - The program recurses with add_last(5, [], NewT).
   -
4. **Base Case (Fourth Call):**

   add_last(5, [], [5]).

   - This is the base case, where the list is empty. The result is [5].
5. **Returning to the Third Call:**

   - The result from the base case is [5].
   - The head 3 is added to this result, so:

     NewT = [3, 5].
6. **Returning to the Second Call:**

   - The result from the third call is [3, 5].
   - The head 2 is added to this result, so:

     NewT = [2, 3, 5].
7. **Returning to the First Call:**

   - The result from the second call is [2, 3, 5].
   - The head 1 is added to this result, so:

     Result = [1, 2, 3, 5].

6. Write a Prolog program to find last item of the list.

Code:

```
last_item([X], X).  % Base case: The last item of a list with one element is that element itself.
last_item([_|T], X) :- last_item(T, X).
% Recursive case: Traverse the list, ignoring the head, until you find the last element.
```

Explanation:

- last_item/2 is a predicate with two arguments:
    1. [X]: A list where X is the single element, which is the last element in that list.
    2. X: The last element to be found in the list.

Base Case:

```
last_item([X], X).
```

- This is the base case. It says that if the list contains only one element ([X]), then that element is the last item in the list.
- **Example**:

  ```
  ?- last_item([3], X).
  X = 3.
  ```

Recursive Case:

```
last_item([_|T], X) :-
    last_item(T, X).
```

- This rule says that if the list has more than one element, ignore the head (_) and recursively call last_item/2 on the tail (T). The recursion continues until the list has only one element (the base case).
- **Example**:

  ```
  ?- last_item([1, 2, 3, 4], X).
  X = 4.
  ```

Recursive Workflow:

Let's go through an example where we want to find the last element of the list [1, 2, 3, 4].

Query:

```
?- last_item([1, 2, 3, 4], X).
```

1. **First Call:**

   ```
   last_item([1, 2, 3, 4], X).
   ```

   - The list has more than one element, so the head (1) is ignored.
   - The program recurses on the tail [2, 3, 4]:

     ```
     last_item([2, 3, 4], X).
     ```

2. **Second Call:**

   last_item([2, 3, 4], X).

   - The head (2) is ignored.
   - The program recurses on the tail [3, 4]:

     last_item([3, 4], X).

3. **Third Call:**

   last_item([3, 4], X).

   - The head (3) is ignored.
   - The program recurses on the tail [4]:

     last_item([4], X).

4. **Base Case (Fourth Call):**

   last_item([4], X).

   - The list now has only one element ([4]), so the base case is triggered.
   - The result is X = 4.
5. **Returning to Previous Calls:**

   - The result X = 4 is passed back to each previous call, so the final result is:

     X = 4.

07. Write a Prolog program to find the nth element of a list.

Code :

```
nth_element(1, [H|_], H).  % Base case: If N is 1, the first element (head) is the result.
nth_element(N, [_|T], X) :-
    N > 1,  % Ensure N is greater than 1.
    N1 is N - 1,  % Decrement N.
    nth_element(N1, T, X).  % Recursively find the nth element in the tail.
```

Explanation:

- nth_element/3 is a predicate with three arguments:
  1. N: The index of the element you want to find in the list.
  2. [H|T]: The list, where H is the head (the first element) and T is the tail (the rest of the list).
  3. X: The element in the nth position.

Recursive Workflow:

Let’s go through an example where we want to find the 3rd element of the list [10, 20, 30, 40].

Query:

?- nth_element(3, [10, 20, 30, 40], X).

1. **First Call:**

   nth_element(3, [10, 20, 30, 40], X).

   - N is 3, which is greater than 1. The head 10 is ignored.
   - N1 is computed as N - 1 = 2.
   - The program recurses on the tail [20, 30, 40]:

     prolog
     Copy code
     nth_element(2, [20, 30, 40], X).

2. **Second Call:**

   nth_element(2, [20, 30, 40], X).

   - N is 2, which is still greater than 1. The head 20 is ignored.
   - N1 is computed as N - 1 = 1.
   - The program recurses on the tail [30, 40]:

     prolog
     Copy code
     nth_element(1, [30, 40], X).

3. **Third Call (Base Case):**

   nth_element(1, [30, 40], X).

   - Now, N is 1, so the base case is triggered.
   - The head of the list is 30, which is the 3rd element.
   - The result is X = 30.

4. **Returning to Previous Calls:**

   - The result X = 30 is passed back to the second and first calls, so the final result is:

     X = 30.

08. Write a Prolog program to check whether a year is a leap year or not.

Code :

```
leap_year(Year) :-
    Year < 0 ->
    (write(Year), write(' your number negative'), nl)  % If the year is negative, display a message.
    ;
    (Year mod 4 =:= 0,
    (Year mod 100 =\= 0 ; Year mod 400 =:= 0)) ->
    (write(Year), write(' is a leap year.'), nl)  % If the year is divisible by 4 and either not divisible by 100 or divisible by 400, it's a leap year.
    ;
    (write(Year), write(' is not a leap year.'), nl).  % Otherwise, it's not a leap year.
```

This Prolog program determines whether a given year is a leap year or not and handles negative numbers as well. The program uses conditional statements to evaluate whether the input year satisfies the rules for being a leap year, and it provides an appropriate message based on the outcome.

Code Breakdown:

prolog
Copy code

```
leap_year(Year) :-
    Year < 0 ->
    (write(Year), write(' your number negative'), nl)  % If the year is negative, display a message.
    ;
    (Year mod 4 =:= 0,
    (Year mod 100 =\= 0 ; Year mod 400 =:= 0)) ->
    (write(Year), write(' is a leap year.'), nl)  % If the year is divisible by 4 and either not divisible by 100 or divisible by 400, it's a leap year.
    ;
    (write(Year), write(' is not a leap year.'), nl).  % Otherwise, it's not a leap year.
```

Explanation:

1. **Base Structure:**
    - The program uses the Prolog conditional (->) and the alternative (;) constructs to create an if-else type logic. It evaluates a series of conditions and prints a message based on the result.

A leap year must satisfy the following conditions:

- The year is divisible by 4 (Year mod 4 =:= 0).
- Either the year is **not divisible by 100** (Year mod 100 =\= 0), or it is **divisible by 400** (Year mod 400 =:= 0).
- If this condition is satisfied, the program prints that the year is a leap year.

09. Write a Prolog program to implement Depth First Search algorithm.

```
% Edge(Node, Neighbor) defines the edges of the graph
edge(a, b).
edge(a, c).
edge(b, d).
edge(b, e).
edge(c, f).
edge(d, g).
edge(e, h).
edge(f, i).
edge(g, j).
edge(h, k).

% dfs(Start, Goal, Path) - Find a path from Start to Goal using DFS
dfs(Start, Goal, Path) :-
    dfs_helper(Start, Goal, [Start], Path).

dfs_helper(Goal, Goal, Visited, Path) :-
    reverse(Visited, Path).

dfs_helper(CurrentNode, Goal, Visited, Path) :-
    edge(CurrentNode, NextNode),
    \+ member(NextNode, Visited),
    dfs_helper(NextNode, Goal, [NextNode|Visited], Path).
```

DFS Search (dfs/3 predicate)

The dfs(Start, Goal, Path) predicate finds a path from the Start node to the Goal node using depth-first search.

- **Parameters:**
  - Start: The starting node for the search.
  - Goal: The node we want to reach.
  - Path: The list of nodes representing the path from Start to Goal.

This predicate calls a helper predicate dfs_helper/4 that does the actual DFS, passing an initial list of visited nodes (which contains only the Start node at the beginning).

3. DFS Helper (dfs_helper/4 predicate)

This is where the DFS logic is implemented. It has three cases:

- **Base Case (Success Case):**

  dfs_helper(Goal, Goal, Visited, Path) :-

  reverse(Visited, Path)

This base case is hit when the current node (Goal) is the same as the goal node, meaning the search has reached the destination. The list of visited nodes (Visited) is reversed (because nodes are prepended during the search) to obtain the correct path from Start to Goal.

Recursive Case (Exploring Neighbors):

```
dfs_helper(CurrentNode, Goal, Visited, Path) :-
  edge(CurrentNode, NextNode),
  \+ member(NextNode, Visited),
  dfs_helper(NextNode, Goal, [NextNode|Visited], Path).
```

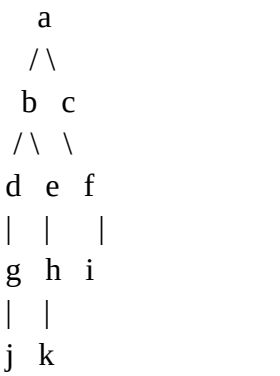
This case is used to explore neighboring nodes. The logic works as follows:

- The predicate edge(CurrentNode, NextNode) checks for an edge between CurrentNode and NextNode.
- The condition \+ member(NextNode, Visited) ensures that the NextNode hasn't been visited already (to avoid cycles).
- If both conditions are satisfied, the search continues by calling dfs_helper/4 recursively, adding NextNode to the list of visited nodes ([NextNode|Visited]).

In essence, this recursively explores the graph in a depth-first manner, backtracking when a dead-end is reached or the Goal is found.

dfs(a, k, Path).

Example:



Step-by-Step Execution of dfs(a, k, Path)

1. **Initial Call:**
   - We start by calling dfs(a, k, Path). This invokes dfs_helper(a, k, [a], Path).
   - Visited = [a] means we have visited node a so far.
2. **Exploring Neighbors of a:**
   - The first edge is edge(a, b), so we move to b.
   - The recursive call becomes dfs_helper(b, k, [b, a], Path).
   - Now, Visited = [b, a] means we have visited nodes b and a.
3. **Exploring Neighbors of b:**
   - The first edge is edge(b, d), so we move to d.
   - The recursive call becomes dfs_helper(d, k, [d, b, a], Path).
   - Now, Visited = [d, b, a] means we have visited nodes d, b, and a.
4. **Exploring Neighbors of d:**

- The first edge is edge(d, g), so we move to g.
- The recursive call becomes dfs_helper(g, k, [g, d, b, a], Path).
- Now, Visited = [g, d, b, a] means we have visited nodes g, d, b, and a.

5. **Exploring Neighbors of g:**
    - The first edge is edge(g, j), so we move to j.
    - The recursive call becomes dfs_helper(j, k, [j, g, d, b, a], Path).
    - Now, Visited = [j, g, d, b, a].
6. **Exploring Neighbors of j:**
    - Node j has no unvisited neighbors (only the g node which is already visited), so we backtrack.
    - We return to node g and continue exploring its neighbors, but there are no more unvisited nodes for g.
    - So, we backtrack again to node d, then b.
7. **Backtracking to b and Exploring Other Neighbors:**
    - Now, we explore the second neighbor of b, which is e.
    - The recursive call becomes dfs_helper(e, k, [e, b, a], Path).
    - Now, Visited = [e, b, a] means we have visited nodes e, b, and a.
8. **Exploring Neighbors of e:**
    - The first edge is edge(e, h), so we move to h.
    - The recursive call becomes dfs_helper(h, k, [h, e, b, a], Path).
    - Now, Visited = [h, e, b, a].
9. **Exploring Neighbors of h:**
    - The first edge is edge(h, k), so we move to k.
    - The recursive call becomes dfs_helper(k, k, [k, h, e, b, a], Path).
    - Now, Visited = [k, h, e, b, a].
10. **Goal Reached:**
    - We have reached the Goal node k, so we hit the base case dfs_helper(k, k, Visited, Path).
    - The list of visited nodes is reversed to get the path from a to k, resulting in Path = [a, b, e, h, k].

Final Output

Path = [a, b, e, h, k].

10.Write a Prolog program to implement Breadth First Search algorithm.

```
% edge(Node, Neighbor) defines the edges of the graph
edge(a, b).
edge(a, c).
edge(b, d).
edge(b, e).
edge(c, f).
edge(d, g).
edge(e, h).
edge(f, i).
edge(g, j).
edge(h, k).

% bfs(Start, Goal, Path) - Find a path from Start to Goal using BFS
bfs(Start, Goal, Path) :-
   bfs_helper([[Start]], Goal, Path).

bfs_helper([[Goal|Rest]|_], Goal, Path) :-
   reverse([Goal|Rest], Path).  % Found the goal, return the path.

bfs_helper([CurrentPath|Paths], Goal, FinalPath) :-
   CurrentPath = [CurrentNode|_],
   findall([NextNode|CurrentPath],
         (edge(CurrentNode, NextNode), \+ member(NextNode, CurrentPath)),
         NewPaths),
   append(Paths, NewPaths, UpdatedPaths),
   bfs_helper(UpdatedPaths, Goal, FinalPath).
```

Breakdown of the Code:

1. **Graph Definition**: The edges of the graph are defined using edge(Node, Neighbor) predicates. For example:

   edge(a, b).

2. edge(a, c).

3. edge(b, d).

4. edge(b, e).

BFS Function:

```
bfs(Start, Goal, Path) :-
   bfs_helper([[Start]], Goal, Path).
```

- bfs(Start, Goal, Path) is the main function that starts the BFS search. It calls the helper function bfs_helper/3 with a list containing the start node, [[Start]].
- Path will be the resulting path from Start to Goal once found.

Helper Function for BFS: The real BFS logic is inside the helper function bfs_helper/3.

Base Case (Goal Found):

```
bfs_helper([[Goal|Rest]|_], Goal, Path) :-
    reverse([Goal|Rest], Path).
```

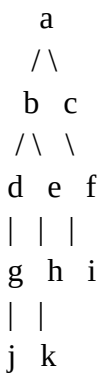
- This clause is triggered when the current path starts with the Goal node ([Goal|Rest]), meaning we have found the goal.
- It constructs the final path by reversing [Goal|Rest] (because paths are built backwards).
- The recursion ends here, returning the Path.

Recursive Case (Expand Neighbors):

```
bfs_helper([CurrentPath|Paths], Goal, FinalPath) :-
    CurrentPath = [CurrentNode|_],
    findall([NextNode|CurrentPath],
        (edge(CurrentNode, NextNode), \+ member(NextNode, CurrentPath)),
        NewPaths),
    append(Paths, NewPaths, UpdatedPaths),
    bfs_helper(UpdatedPaths, Goal, FinalPath).
```

- **CurrentPath = [CurrentNode|_]**: Takes the first path from the list of paths to explore (CurrentPath) and extracts the CurrentNode.
- **findall([NextNode|CurrentPath], ...)**: This line expands the CurrentNode by finding all neighbors (NextNode) of CurrentNode that haven't been visited yet (i.e., \+ member(NextNode, CurrentPath) ensures the next node hasn't already been visited).
- For each valid neighbor (NextNode), a new path is created by adding NextNode to the current path ([NextNode|CurrentPath]).
- **append(Paths, NewPaths, UpdatedPaths)**: Appends the new paths (NewPaths) generated from the current node to the rest of the paths (Paths) that haven't been explored yet. This ensures that BFS explores all nodes level by level.
- **Recursive call**: The function then calls itself with the updated list of paths to explore (UpdatedPaths), repeating the process until the goal is found.

Graph Representation:



Step-by-Step Execution of bfs(a, e, Path):

1. **Initial Call:**
    - We start with the call bfs(a, e, Path). This invokes bfs_helper([[a]], e, Path).
    - The [[a]] represents the current paths to explore, starting from a.
2. **First Iteration (bfs_helper/3):**
    - The current path is CurrentPath = [a] (starting at node a).
    - We now look at the neighbors of a using the findall/3 predicate:
        - edge(a, b) gives the path [b, a].
        - edge(a, c) gives the path [c, a].
    - The new paths are NewPaths = [[b, a], [c, a]].
    - We append the new paths to the remaining paths, resulting in UpdatedPaths = [[b, a], [c, a]].
    - Recursive call becomes bfs_helper([[b, a], [c, a]], e, Path).
3. **Second Iteration (bfs_helper/3):**
    - The current path is CurrentPath = [b, a].
    - We now look at the neighbors of b:
        - edge(b, d) gives the path [d, b, a].
        - edge(b, e) gives the path [e, b, a].
    - The new paths are NewPaths = [[d, b, a], [e, b, a]].
    - Append these new paths to the remaining paths [[c, a]], so UpdatedPaths = [[c, a], [d, b, a], [e, b, a]].
    - Recursive call becomes bfs_helper([[c, a], [d, b, a], [e, b, a]], e, Path).
4. **Third Iteration (Goal Found):**
    - The current path is CurrentPath = [e, b, a].
    - Since CurrentNode = e matches the goal node e, we have found the goal.
    - The base case is triggered, and we reverse the path [e, b, a] to get the final result Path = [a, b, e].