Experiment- 4

Problem Statement

Implement a city database using unordered lists . each database reocrd contains the name of city (a string of arbitary length) and the coordinates of the city expresses as interger x and y coordinates . your program should allow following instruction .

a) insert record

b) delete record by name or coordinate

c) search a record by name or coordinate

d) Point all records within a given distance of specific point

Implement using Array and Linked List

Linked Lists:

Solution

The implementation of the city database using a linked list is a way to store and manage a collection of city records.

The City class represents each record and contains the name of the city and its x and y coordinates.

The CityDatabase class is the main data structure used to manage the city records. It is implemented using a linked list where each node contains a City object and a reference to the next node in the list.

The insert() method is used to add a new city record to the database. The new record is inserted at the beginning of the linked list.

The delete_by_name() and delete_by_coordinate() methods are used to remove a city record from the database. These methods traverse the linked list and remove the node that contains the city record with the matching name or coordinates.

The search_by_name() and search_by_coordinate() methods are used to find a city record in the database. These methods traverse the linked list and return the node that contains the city record with the matching name or coordinates.

The points_within_distance() method is used to find all city records within a given distance of a specific point. This method traverses the linked list and calculates the distance between each city record and the given point using the distance formula. If the distance is less than or equal to the given distance, the city record is added to a list of results.

Overall, the linked list solution is a simple and efficient way to implement a city database. The time complexity of the insert, delete, and search operations is O(n), where n is the number of city records in the database. The time complexity of the points_within_distance() operation is also O(n), but it depends on the size of the results list, which can be smaller than n

Code Implementation Linked List:

- Basic Linked Lists and dependencies

```
import math

import time

class City:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x
        self.y = y
        self.next = None
```

- City Data Base and Operations

```
class CityDatabase:
    def __init__(self):
        self.head = None

    def insert(self, name, x, y):
        new_city = City(name, x, y)
        new_city.next = self.head
        self.head = new_city

    def delete_by_name(self, name):
        if self.head is None:
            return
        if self.head.name == name:
            self.head = self.head.next
            return
        curr = self.head
        while curr.next is not None:
            if curr.next.name == name:
                curr.next = curr.next.next
                return
            curr = curr.next

    def delete_by_coordinate(self, x, y):
        if self.head is None:
            return
        if self.head.x == x and self.head.y == y:
            self.head = self.head.next
            return
        curr = self.head
        while curr.next is not None:
            if curr.next.x == x and curr.next.y == y:
                curr.next = curr.next.next
                return
            curr = curr.next

    def search_by_name(self, name):
        curr = self.head
        while curr is not None:
            if curr.name == name:
                return curr
            curr = curr.next
        return None

    def search_by_coordinate(self, x, y):
        curr = self.head
        while curr is not None:
            if curr.x == x and curr.y == y:
                return curr
            curr = curr.next
        return None

    def points_within_distance(self, x, y, distance):
        results = []
        curr = self.head
        while curr is not None:
            if math.sqrt((curr.x - x) ** 2 + (curr.y - y) ** 2) <= distance:
                results.append(curr)
            curr = curr.next
        return results
```

Time Complexity

The runtime complexity of the operations on this linked list implementation of a city database are as follows:

Insert: O(1) - since we are inserting at the head of the linked list, the time complexity is constant.

Delete by name: O(n) - in the worst case, we may have to traverse the entire linked list to find the node with the specified name.

Delete by coordinate: O(n) - in the worst case, we may have to traverse the entire linked list to find the node with the specified coordinate.

Search by name: O(n) - in the worst case, we may have to traverse the entire linked list to find the node with the specified name.

Search by coordinate: O(n) - in the worst case, we may have to traverse the entire linked list to find the node with the specified coordinate.

Points within distance: O(n) - we have to traverse the entire linked list and perform a distance calculation on each node to determine if it is within the specified distance.

- Interface

```
# Example usage
if __name__ == '__main__':
    db = CityDatabase()

    # Insert records
    start = time.time()
    db.insert("Los Angeles", 34, -118)
    db.insert("New York", 40, -74)
    db.insert("Chicago", 41, -87)
    db.insert("Houston", 29, -95)
    db.insert("Phoenix", 33, -112)
    end = time.time()
    print("Inserted records in", end - start, "seconds")

    # Search by name
    start = time.time()
    city = db.search_by_name("Chicago")
    end = time.time()
    if city is not None:
        print("Found", city.name, "at", city.x, ",", city.y, "in", end -
start, "seconds")
    else:
        print("City not found")

    # Search by coordinate
    start = time.time()
    city = db.search_by_coordinate(40, -74)
    end = time.time()
    if city is not None:
        print("Found", city.name, "at", city.x, ",", city.y, "in", end -
start, "seconds")
    else:
        print("City not found")

    # Points within distance
    start = time.time()
    cities = db.points_within_distance(40, -74, 500)
    end = time.time()
    if len(cities) > 0:
        print("Cities within distance:")
    for city in cities:
        print(f"{city.name} ({city.x}, {city.y})")
    else:
        print("No cities found within distance.")
    print(f"Time taken: {end - start} seconds.")
```

Output:

```
Inserted records in 0.0 seconds

Found Chicago at 41 , -87 in 0.0 seconds

Found New York at 40 , -74 in 0.0 seconds

Cities within distance:

Phoenix (33, -112)

Houston (29, -95)

Chicago (41, -87)

New York (40, -74)

Los Angeles (34, -118)

No cities found within distance.

Time taken: 0.0 seconds.
```

Code Implementation Array:

- Basic Linked Lists and dependencies

```
import math

import time

class City:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x
        self.y = y
        self.next = None
```

- City Data Base and Operations

```
class CityDatabase:
    def __init__(self):
        self.head = None

    def insert(self, name, x, y):
        new_city = City(name, x, y)
        new_city.next = self.head
        self.head = new_city

    def delete_by_name(self, name):
        if self.head is None:
            return
        if self.head.name == name:
            self.head = self.head.next
            return
        curr = self.head
        while curr.next is not None:
            if curr.next.name == name:
                curr.next = curr.next.next
                return
            curr = curr.next

    def delete_by_coordinate(self, x, y):
        if self.head is None:
            return
        if self.head.x == x and self.head.y == y:
            self.head = self.head.next
            return
        curr = self.head
        while curr.next is not None:
            if curr.next.x == x and curr.next.y == y:
                curr.next = curr.next.next
                return
            curr = curr.next

    def search_by_name(self, name):
        curr = self.head
        while curr is not None:
            if curr.name == name:
                return curr
            curr = curr.next
        return None

    def search_by_coordinate(self, x, y):
        curr = self.head
        while curr is not None:
            if curr.x == x and curr.y == y:
                return curr
            curr = curr.next
        return None

    def points_within_distance(self, x, y, distance):
        results = []
        curr = self.head
        while curr is not None:
            if math.sqrt((curr.x - x) ** 2 + (curr.y - y) ** 2) <= distance:
                results.append(curr)
            curr = curr.next
        return results
```

Insert: O(1) - Adding an element to the end of an array has a constant time complexity.

Delete by name or coordinate: O(n) - In the worst case, we have to traverse the entire array to find the element to delete, giving us a time complexity of O(n).

Search by name or coordinate: O(n) - In the worst case, we have to traverse the entire array to find the desired element, giving us a time complexity of O(n).

Points within a given distance of a specific point: O(n) - In the worst case, we have to traverse the entire array to find all the points within the given distance, giving us a time complexity of O(n).

- Interface

```
# Example usage
if __name__ == '__main__':
    db = CityDatabase()

    # Insert records
    start = time.time()
    db.insert("Los Angeles", 34, -118)
    db.insert("New York", 40, -74)
    db.insert("Chicago", 41, -87)
    db.insert("Houston", 29, -95)
    db.insert("Phoenix", 33, -112)
    end = time.time()
    print("Inserted records in", end - start, "seconds")

    # Search by name
    start = time.time()
    city = db.search_by_name("Chicago")
    end = time.time()
    if city is not None:
        print("Found", city.name, "at", city.x, ",", city.y, "in", end -
start, "seconds")
    else:
        print("City not found")

    # Search by coordinate
    start = time.time()
    city = db.search_by_coordinate(40, -74)
    end = time.time()
    if city is not None:
        print("Found", city.name, "at", city.x, ",", city.y, "in", end -
start, "seconds")
    else:
        print("City not found")

    # Points within distance
    start = time.time()
    cities = db.points_within_distance(40, -74, 500)
    end = time.time()
    if len(cities) > 0:
        print("Cities within distance:")
    for city in cities:
        print(f"{city.name} ({city.x}, {city.y})")
    else:
        print("No cities found within distance.")
    print(f"Time taken: {end - start} seconds.")
```

Output:

```
Inserted records in 0.0 seconds

Found Chicago at 41 , -87 in 0.0 seconds

Found New York at 40 , -74 in 0.0 seconds

Cities within distance:

Los Angeles (34, -118)

New York (40, -74)

Chicago (41, -87)

Houston (29, -95)Phoenix (33, -112)

No cities found within distance.

Time taken: 0.0 seconds.
```

Questions

Advantages and Disadvantages of Implementing a City Database using Unordered Lists:

Category	Array Implementation	Linked List Implementation
Memory Usage	Fixed memory allocation required, regardless of the number of records in the database	Memory usage is flexible, and only increases as more records are added
Insertion	Insertion can be slow, as all elements after the insertion point must be shifted to make space	Insertion is relatively fast, as only one new node needs to be created and inserted
Deletion	Deletion can be slow, as all elements after the deleted element must be shifted to fill the gap	Deletion is relatively fast, as only the pointer to the deleted node needs to be updated
Search	Search is relatively fast if the records are sorted, but can be slow otherwise	Search is slower than array implementation, as linked lists do not provide random access
Point Query	Point queries are relatively fast, as the array allows for direct access to the coordinates	Point queries can be slower, as each node must be traversed to compare coordinates

Category	Array Implementation	Linked List Implementation
Space Efficiency	Arrays can be less space efficient than linked lists, as they require fixed memory allocation regardless of the number of records	Linked lists can be more space efficient than arrays, as they only require memory for the records that are actually present

general, implementing a city database using an array may be faster for searching and point queries, but slower for insertion and deletion. On the other hand, implementing a city database using a linked list may be slower for searching and point queries, but faster for insertion and deletion. However, the choice of implementation ultimately depends on the specific requirements of the application and the constraints of the system it will run on.

Would you be storing records on list in alphabetical order by city name, will speed of operation be affected?

Yes, if the records are stored in alphabetical order by city name, the speed of certain operations may be affected, depending on the implementation.

If the database is implemented as an array, searching for a record by city name would be faster if the records are stored in alphabetical order, as a binary search can be used to quickly locate the record. However, insertion and deletion would be slower, as all the elements after the insertion or deletion point would need to be shifted to maintain the alphabetical order.

If the database is implemented as a linked list, searching for a record by city name would also be faster if the records are stored in alphabetical order, as the linked list can be traversed until the correct record is found. However, insertion and deletion would be faster than with an array implementation, as only the pointers to adjacent nodes need to be updated to maintain the alphabetical order.

In summary, storing the records in alphabetical order may improve the speed of searching for records by city name, but may decrease the speed of insertion and deletion operations, depending on the implementation.

Would you be keeping the list in alphabetical order slow any of operation?

Yes, keeping the list in alphabetical order would slow down some operations, depending on the specific implementation.

If the database is implemented using an array, inserting or deleting a record in alphabetical order would require shifting all elements in the array after the insertion or deletion point, which could be a time-consuming operation, especially if the array is large. Searching for a record by name in an alphabetically ordered array, however, can be done more quickly using binary search, as the array is sorted.

If the database is implemented using a linked list, inserting or deleting a record in alphabetical order would also require traversing the linked list to find the correct position for the new record or the record to be deleted. However, because linked lists are more flexible in terms of memory usage than arrays, insertion, and deletion operations are generally faster than with arrays. Searching for a record by name in an alphabetically ordered linked list is slower than searching in an alphabetically ordered array using binary search, as the linked list must be traversed in sequence.

Overall, keeping the list in alphabetical order may slow down certain operations, such as insertion and deletion, but it may also speed up searching for a record by name. The impact on performance will depend on the specific implementation and the size of the database.