MultiMaps/MultiMaps.Core/gpt/combined_source_files

```

=== File: ../Internal/Bucket.cs ===
namespace MultiMaps.Core.Internal;

internal class Bucket<TKey, TValue>
{
    public TKey Key { get; }
    public HashSet<TValue> Values { get; }

    public Bucket(TKey key)
    {
        Key = key;
        Values = new HashSet<TValue>();
    }
}

=== File: ../Internal/BucketComparer.cs ===
namespace MultiMaps.Core.Internal;

public class BucketComparer<TKey> : IEqualityComparer<TKey>
{
    private readonly int _bucketCount;
    private readonly IEqualityComparer<TKey> _innerComparer;

    public BucketComparer(int bucketCount)
        : this(bucketCount, EqualityComparer<TKey>.Default)
    {
    }

    public BucketComparer(
        int bucketCount,
        IEqualityComparer<TKey> innerComparer)
    {
        if (bucketCount <= 0)
        {
            throw new ArgumentOutOfRangeException(
                nameof(bucketCount),
                "Bucket count must be positive");
        }

        _bucketCount = bucketCount;
        _innerComparer = innerComparer
            ?? throw new ArgumentNullException(nameof(innerComparer));
    }

    public bool Equals(TKey? x, TKey? y)
    {
        return _innerComparer.Equals(x, y);
    }

    public int GetHashCode(TKey obj)
    {
        if (obj == null)
            return 0;

        var originalHash = _innerComparer.GetHashCode(obj);

        return Math.Abs(originalHash % _bucketCount);
    }

    public int GetBucketIndex(TKey key)
    {
        return GetHashCode(key);
    }

    public int BucketCount => _bucketCount;
}

=== File: ../Internal/FnvHashStrategy.cs ===
namespace MultiMaps.Core.Internal;

public class FnvHashStrategy<TKey> : IEqualityComparer<TKey>
{
    private const uint FNV_PRIME = 16777619;
    private const uint FNV_OFFSET_BASIS = 2166136261;
    private readonly IEqualityComparer<TKey> _innerComparer;

    public FnvHashStrategy()
        : this(EqualityComparer<TKey>.Default)
    {
    }

    public FnvHashStrategy(IEqualityComparer<TKey> innerComparer)
    {
        _innerComparer = innerComparer
            ?? throw new ArgumentNullException(nameof(innerComparer));
    }

    public bool Equals(TKey? x, TKey? y)
    {
        return _innerComparer.Equals(x, y);
    }

    public int GetHashCode(TKey obj)
    {
        if (obj == null)
            return 0;

```

```csharp
        var originalHash = _innerComparer.GetHashCode(obj);
        uint hash = FNV_OFFSET_BASIS;

        var bytes = BitConverter.GetBytes(originalHash);
        foreach (var b in bytes)
        {
            hash ^= b;
            hash *= FNV_PRIME;
        }

        return (int)hash;
    }
}

public class MurmurHashStrategy<TKey> : IEqualityComparer<TKey>
{
    private const uint SEED = 0x9747b28c;
    private const uint M = 0x5bd1e995;
    private const int R = 24;
    private readonly IEqualityComparer<TKey> _innerComparer;

    public MurmurHashStrategy()
        : this(EqualityComparer<TKey>.Default)
    {
    }

    public MurmurHashStrategy(IEqualityComparer<TKey> innerComparer)
    {
        _innerComparer = innerComparer
            ?? throw new ArgumentNullException(nameof(innerComparer));
    }

    public bool Equals(TKey? x, TKey? y)
    {
        return _innerComparer.Equals(x, y);
    }

    public int GetHashCode(TKey obj)
    {
        if (obj == null)
            return 0;

        var originalHash = _innerComparer.GetHashCode(obj);
        var bytes = BitConverter.GetBytes(originalHash);
        uint h = SEED ^ (uint)bytes.Length;

        foreach (var b in bytes)
        {
            uint k = b;
            k *= M;
            k ^= k >> R;
            k *= M;

            h *= M;
            h ^= k;
        }

        h ^= h >> 13;
        h *= M;
        h ^= h >> 15;

        return (int)h;
    }
}

=== File: ../Internal/HashStrategyFactory.cs ===
namespace MultiMaps.Core;

public enum HashingAlgorithm
{
    Default,
    FowlerNollVo,
    Murmur
}

public static class HashingStrategies
{
    public static IEqualityComparer<TKey> Create<TKey>(
        HashingAlgorithm algorithm)
    {
        return algorithm switch
        {
            HashingAlgorithm.FowlerNollVo
                => new Internal.FnvHashStrategy<TKey>(),

            HashingAlgorithm.Murmur
                => new Internal.MurmurHashStrategy<TKey>(),

            _ => EqualityComparer<TKey>.Default
        };
    }

    public static IEqualityComparer<TKey> CreateBucketed<TKey>(
        int bucketCount,
        HashingAlgorithm algorithm = HashingAlgorithm.Default)
    {
        var baseComparer = Create<TKey>(algorithm);

        return new Internal.BucketComparer<TKey>(
            bucketCount,
            baseComparer);
    }
}

=== File: ../Internal/MapEnumerator.cs ===
using System.Collections;

namespace MultiMaps.Core.Internal;

internal class MapEnumerator<TKey, TValue>
    : IEnumerator<KeyValuePair<TKey, ISet<TValue>>>
{
    private readonly List<Bucket<TKey, TValue>> _buckets;
    private int _currentIndex;
    private KeyValuePair<TKey, ISet<TValue>> _current;

    public MapEnumerator(List<Bucket<TKey, TValue>> buckets)
    {
        _buckets = buckets;
        _currentIndex = -1;
        _current = default;
    }

    public KeyValuePair<TKey, ISet<TValue>> Current => _current;

    object IEnumerator.Current => Current;

    public void Dispose()
    {
        // No unmanaged resources to dispose
    }

    public bool MoveNext()
    {
        if (_currentIndex < _buckets.Count - 1)
        {
            _currentIndex++;

            var bucket = _buckets[_currentIndex];
            _current = new KeyValuePair<TKey, ISet<TValue>>(
                bucket.Key,
                bucket.Values);

            return true;
        }

        return false;
    }

    public void Reset()
    {
        _currentIndex = -1;
        _current = default;
    }
}

=== File: ../MultiMaps.Core.csproj ===
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>


=== File: ../OneToManyMap.cs ===
using System.Collections;
using MultiMaps.Core.Internal;

namespace MultiMaps.Core;

public class OneToManyMap<TKey, TValue>
    : IEnumerable<KeyValuePair<TKey, ISet<TValue>>>
{
    private readonly List<Bucket<TKey, TValue>> _buckets;
    private readonly IEqualityComparer<TKey> _comparer;

    public OneToManyMap()
        : this(EqualityComparer<TKey>.Default)
    {
    }

    public OneToManyMap(IEqualityComparer<TKey> comparer)
    {
        _buckets = new List<Bucket<TKey, TValue>>();
        _comparer = comparer
            ?? throw new ArgumentNullException(nameof(comparer));
    }

    public ISet<TValue> this[TKey key]
    {
        get
        {
            var bucket = FindBucket(key)
                ?? throw new KeyNotFoundException(
                    $"The key '{key}' was not found.");

```

```csharp
            return bucket.Values;
        }
    }

    public int Count => _buckets.Count;

    public ICollection<TKey> Keys =>
        _buckets.Select(b => b.Key).ToList();

    public ICollection<ISet<TValue>> Values =>
        _buckets.Select(b => (ISet<TValue>)b.Values).ToList();

    public void Add(TKey key, TValue value)
    {
        var bucket = FindBucket(key);
        if (bucket == null)
        {
            bucket = new Bucket<TKey, TValue>(key);
            _buckets.Add(bucket);
        }
        bucket.Values.Add(value);
    }

    public void AddRange(TKey key, IEnumerable<TValue> values)
    {
        if (values == null)
            throw new ArgumentNullException(nameof(values));

        var bucket = FindBucket(key);
        if (bucket == null)
        {
            bucket = new Bucket<TKey, TValue>(key);
            _buckets.Add(bucket);
        }

        foreach (var value in values)
            bucket.Values.Add(value);
    }

    public bool Remove(TKey key)
    {
        for (int i = 0; i < _buckets.Count; i++)
        {
            if (_comparer.Equals(_buckets[i].Key, key))
            {
                _buckets.RemoveAt(i);
                return true;
            }
        }
        return false;
    }

    public bool RemoveValue(TKey key, TValue value)
    {
        var bucket = FindBucket(key);
        return bucket != null && bucket.Values.Remove(value);
    }

    public bool ContainsKey(TKey key) => FindBucket(key) != null;

    public bool ContainsValue(TKey key, TValue value)
    {
        var bucket = FindBucket(key);
        return bucket != null && bucket.Values.Contains(value);
    }

    public bool TryGetValues(TKey key, out ISet<TValue> values)
    {
        var bucket = FindBucket(key);
        if (bucket != null)
        {
            values = bucket.Values;
            return true;
        }
        values = new HashSet<TValue>();
        return false;
    }

    public void Clear() => _buckets.Clear();

    private Bucket<TKey, TValue>? FindBucket(TKey key)
    {
        if (key == null)
            throw new ArgumentNullException(nameof(key));

        return _buckets.FirstOrDefault(b =>
            _comparer.Equals(b.Key, key));
    }

    public IEnumerator<KeyValuePair<TKey, ISet<TValue>>> GetEnumerator()
    {
        return new MapEnumerator<TKey, TValue>(_buckets);
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    public ISet<TValue> GetOrCreate(TKey key)
    {
        var bucket = FindBucket(key);
        if (bucket == null)
        {
            bucket = new Bucket<TKey, TValue>(key);
            _buckets.Add(bucket);
        }
        return bucket.Values;
    }
}

```