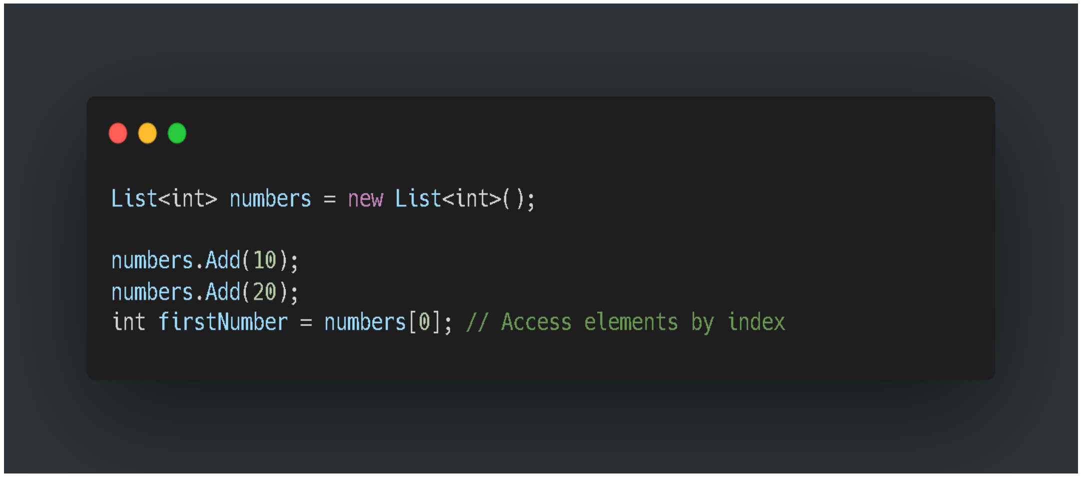
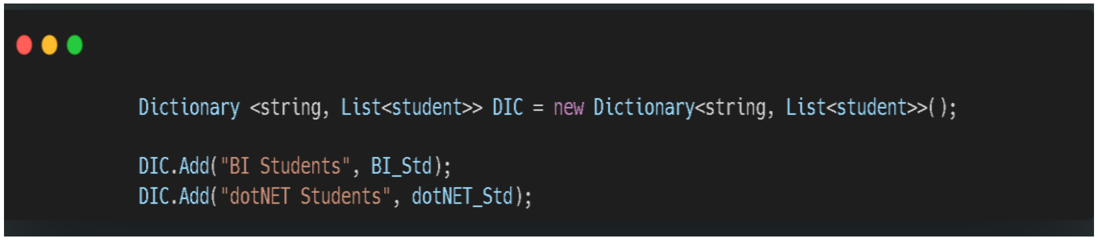
- ***Generic In C#***
    - *Generics are a way to create reusable code that can work with different types of data.*
    - *Generic types are declared using angle brackets (<>).*
    - *can be any type that is supported by the .NET.*
    - *Generic types can be used to create classes, structures, interfaces, and methods.*
    - *Generics provide compile-time type checking, ensuring type safety and reducing runtime errors.*
    - *Code Reusability: Generics enable you to write generic algorithms and data structures that can be used with different data types, promoting code reusability.*
    - *improved performance: Generics avoid boxing and unboxing operations for value types.*
- ***Generic.Collection.***
    - **List<T>**
    - *List<T> is a dynamic array that can grow or shrink in size.*
    - *It allows you to store elements of a specific data type T.*
    - *Provides methods to add, remove, access elements, and more.*



```
List<int> numbers = new List<int>();

numbers.Add(10);
numbers.Add(20);
int firstNumber = numbers[0]; // Access elements by index
```

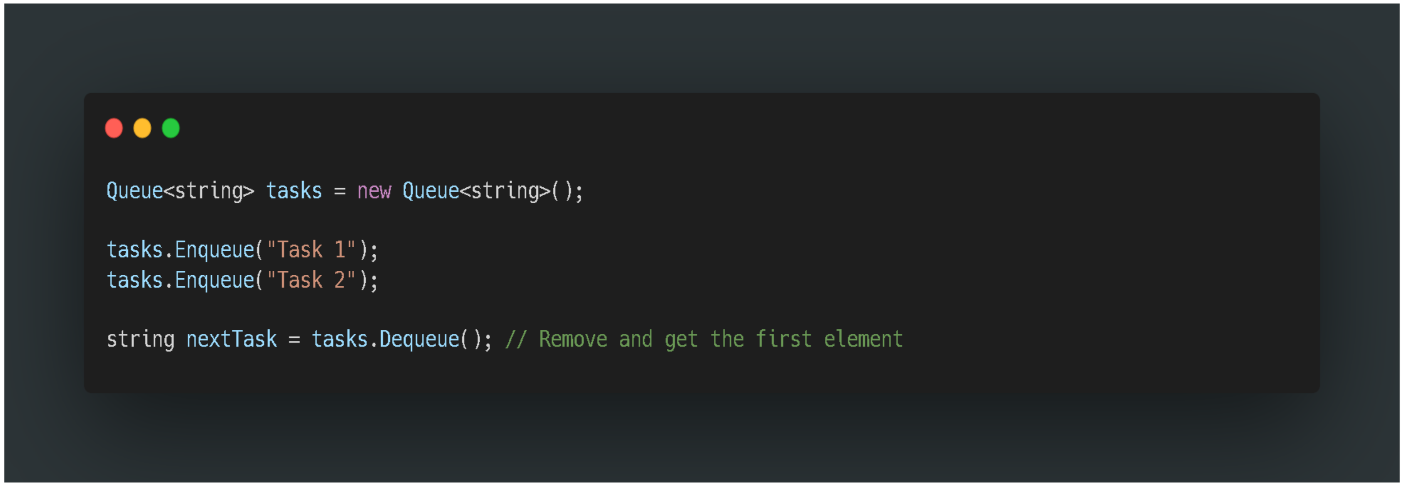
- **Dictionary<TKey, TValue>**
- *Dictionary<TKey, TValue> is a collection of key-value pairs.*
- *It allows you to store elements with unique keys of type TKey and corresponding values of type TValue.*
- *Provides methods to add, remove, access elements, and more using the keys.*



```
Dictionary <string, List<student>> DIC = new Dictionary<string, List<student>>();

DIC.Add("BI Students", BI_Std);
DIC.Add("dotNET Students", dotNET_Std);
```

- **Queue<T>**
- *Queue<T> is a first-in-first-out (FIFO) collection.*
- *where elements are added at the end and removed from the beginning..*
- *It allows you to store elements of type T.*
- *Provides methods to enqueue (add), dequeue (remove), and access elements.*

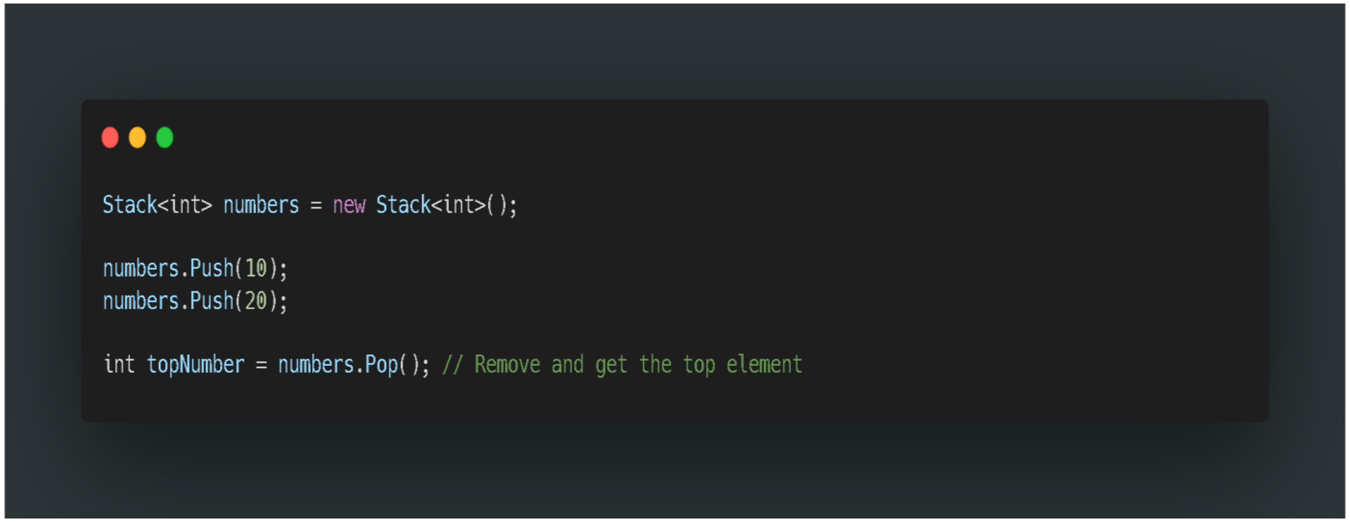


```
Queue<string> tasks = new Queue<string>();

tasks.Enqueue("Task 1");
tasks.Enqueue("Task 2");

string nextTask = tasks.Dequeue(); // Remove and get the first element
```

- **Stack<T>**
- *Stack<T> is a last-in-first-out (LIFO) collection.*
- *where elements are added and removed from the top (end) of the stack.*
- *It allows you to store elements of type T.*
- *Provides methods to push (add), pop (remove), and access elements.*

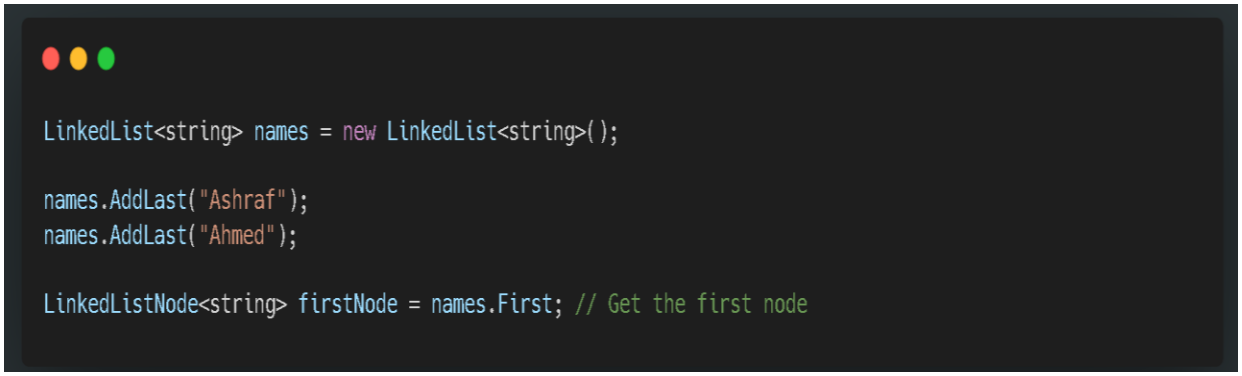


```
Stack<int> numbers = new Stack<int>();

numbers.Push(10);
numbers.Push(20);

int topNumber = numbers.Pop(); // Remove and get the top element
```

- **LinkedList<T>**
- *LinkedList<T> is a doubly linked list collection that allows you to store elements of type T.*
- *Provides methods to add, remove, and access elements efficiently.*



```
LinkedList<string> names = new LinkedList<string>();

names.AddLast("Ashraf");
names.AddLast("Ahmed");

LinkedListNode<string> firstNode = names.First; // Get the first node
```

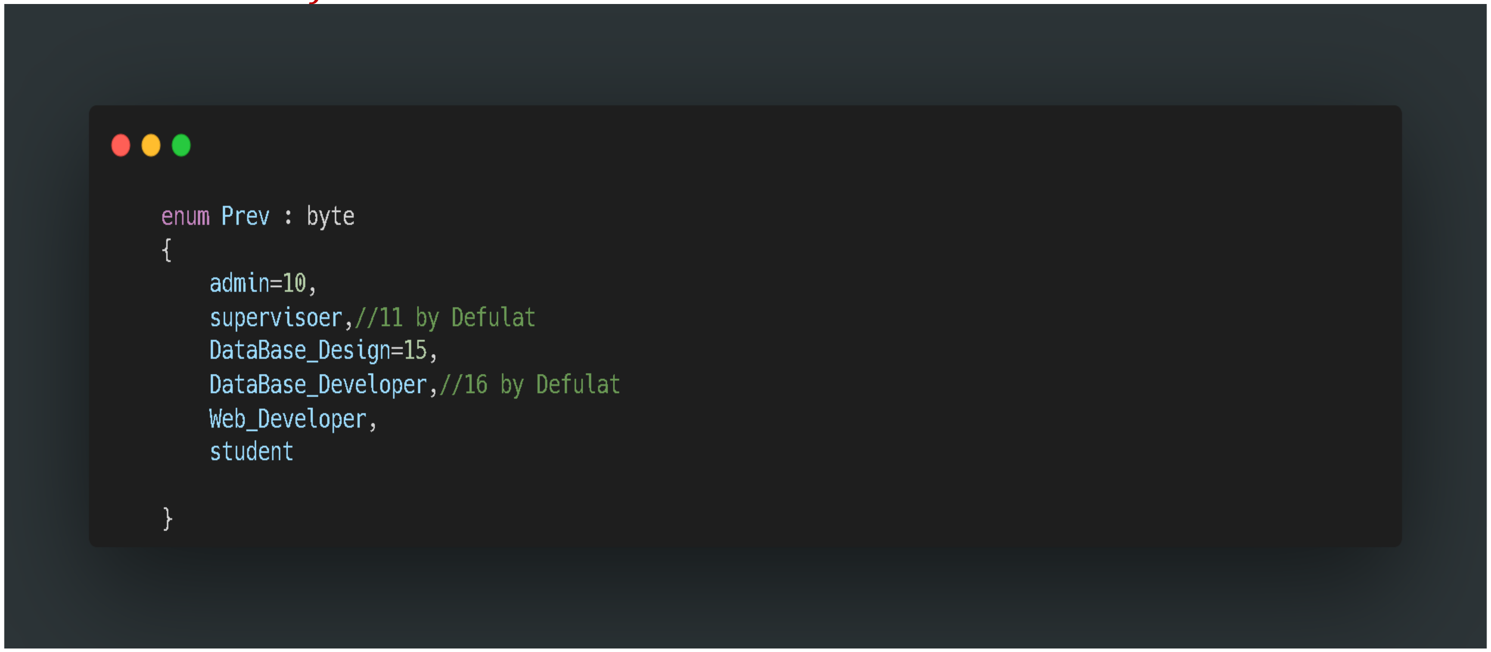
Feature	*List*	*LinkedList*
Pros	• *Efficient for accessing elements by index*	• *Efficient for inserting and removing elements in the middle*
Cons	• *Inefficient for inserting and removing elements in the middle*	• *Less efficient for accessing elements by index*

Feature	*List*	*LinkedList*
Storage	• *Dynamic array*	• *Doubly linked list*
Access by index	• *Efficient*	• *Inefficient*
Insert/remove in the middle	• *Inefficient*	• *Efficient*
Memory usage	• *More efficient*	• *Less efficient*

`Object` class:

  - *The `Object` class is the base class for all other classes in C#.*
  - *Every class implicitly or explicitly inherits from the `Object` class.*
  - *The default implementation of `Equals` in the `Object` class performs reference equality comparison.*
  - *Boxing is the process of converting a value type to a reference type (`Object`).*
  - *unboxing is the reverse process of converting a reference type (`Object`) To value type.*
- *Common Members:*
  - *`Equals`: Compares two objects for value equality.*
  - *`GetHashCode`: Returns a hash code value for the object.*
  - *`ToString`: Returns a string representation of the object.*
  - *`GetType`: Returns the runtime type of the object.*

- ***Enumerations (Enum)***
  - *Enums are a type of value type in C# (means that they are stored on the stack).*
  - *Enums are immutable -> means that their values cannot be changed after they are created.*
  - *Enum members are named constants that represent specific values within the enum type.*
  - *(To make your code more readable and maintainable.)*
  - *Enums can be used in switch statements.*
  - *Enums can be explicitly converted to strings (useful for displaying the value of an enum).*
  - *Enums can be used to implement enumerations.*
  - *Inheritance from*



```
enum Prev : byte
{
    admin=10,
    supervisoer,//11 by Defulat
    DataBase_Design=15,
    DataBase_Developer,//16 by Defulat
    Web_Developer,
    student

}
```

- ***Interfaces in C#***
  - *Allow you to specify a set of method and property signatures without providing implementation details.*
  - *Inside the interface, you can define method signatures, property declarations, events, and indexers, but you cannot provide implementation details.*
  - *Classes that implement an interface must provide implementations for all the members defined in the interface.*
  - *A class can implement multiple interfaces, allowing it to adhere to multiple contracts.*
  - *Interfaces can inherit from other interfaces.*
  - *Interfaces provide a way to achieve abstraction and polymorphism in C#.*
  - *Interfaces play a crucial role in achieving loose coupling between components in object-oriented programming.*
  - *good design principles like separation of concerns and facilitate code reuse and maintainability.*