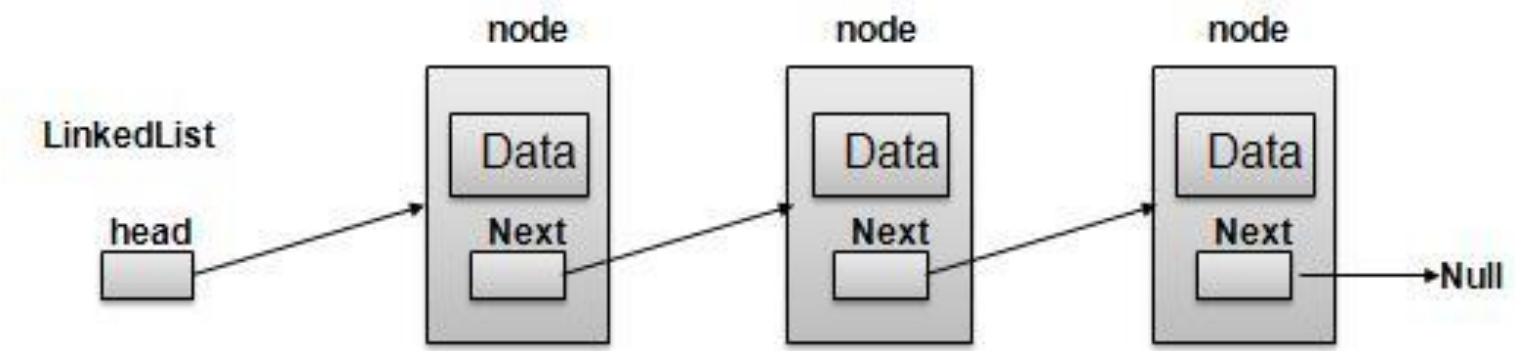
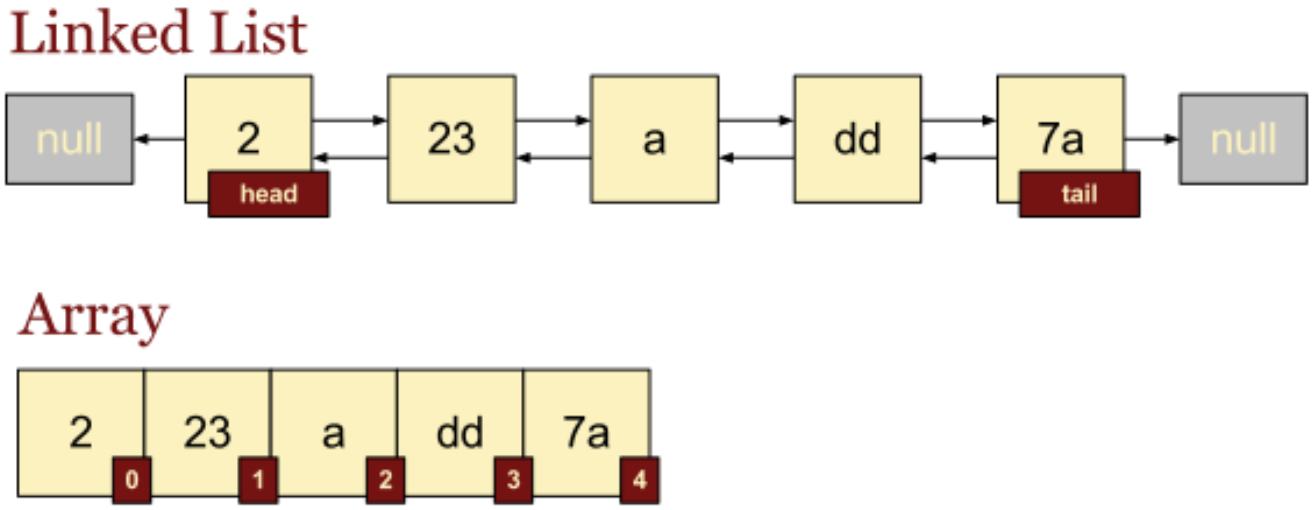
Linked List

Software Entwicklung



Arrays vs. Linked List

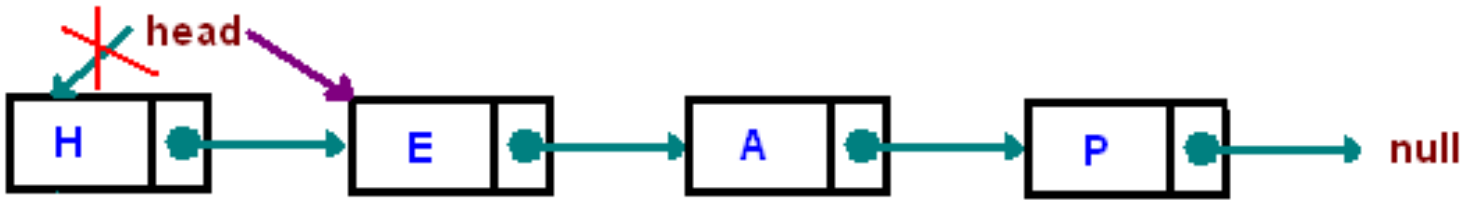
- arrays items are defined by their indices
- linked list item contains a pointer to its predecessor and his successor



Navigate throw Lists

- Set the head at the second element:

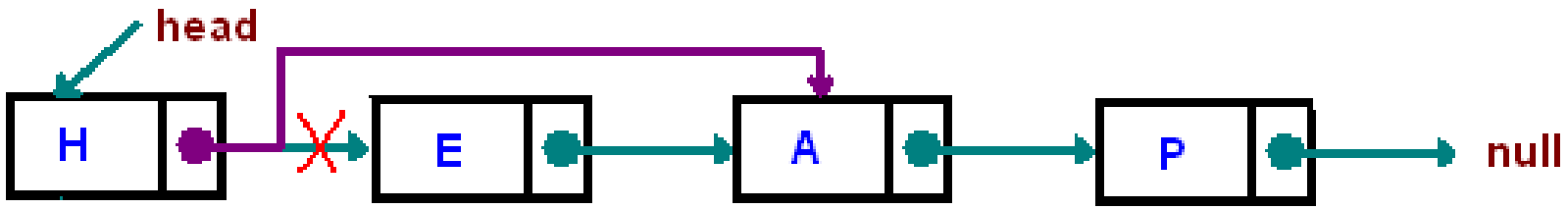
```
head = head.next;
```



Navigation with a Reference

- Remove the second element from the list

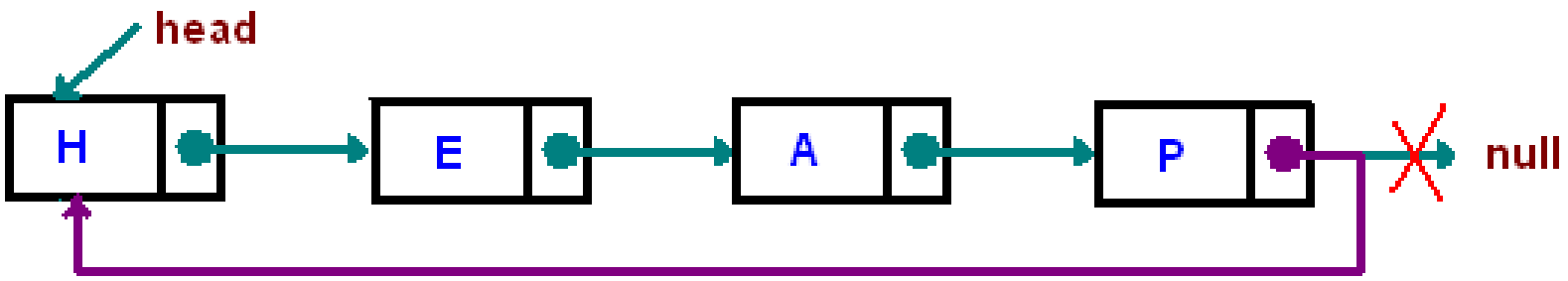
```
head.next = head.next.next;
```



Navigation with a Reference

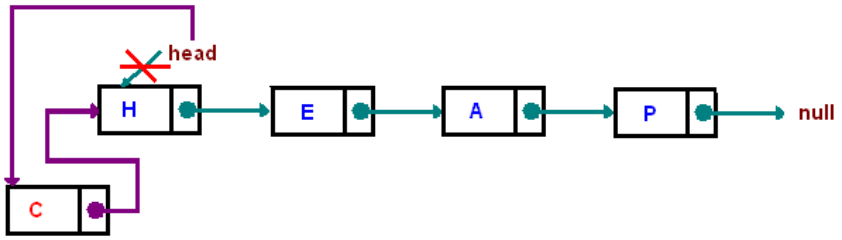
- Listenende auf erstes Listenelement zeigen lassen:

```
head.next.next.next.next = head;
```

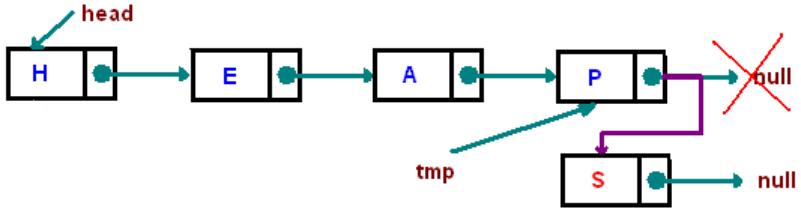


Insert Methods

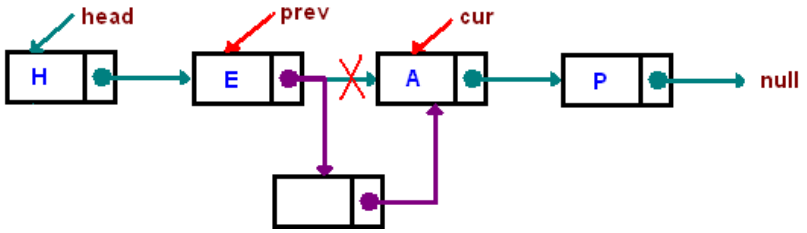
- Insert Front



- Insert Last

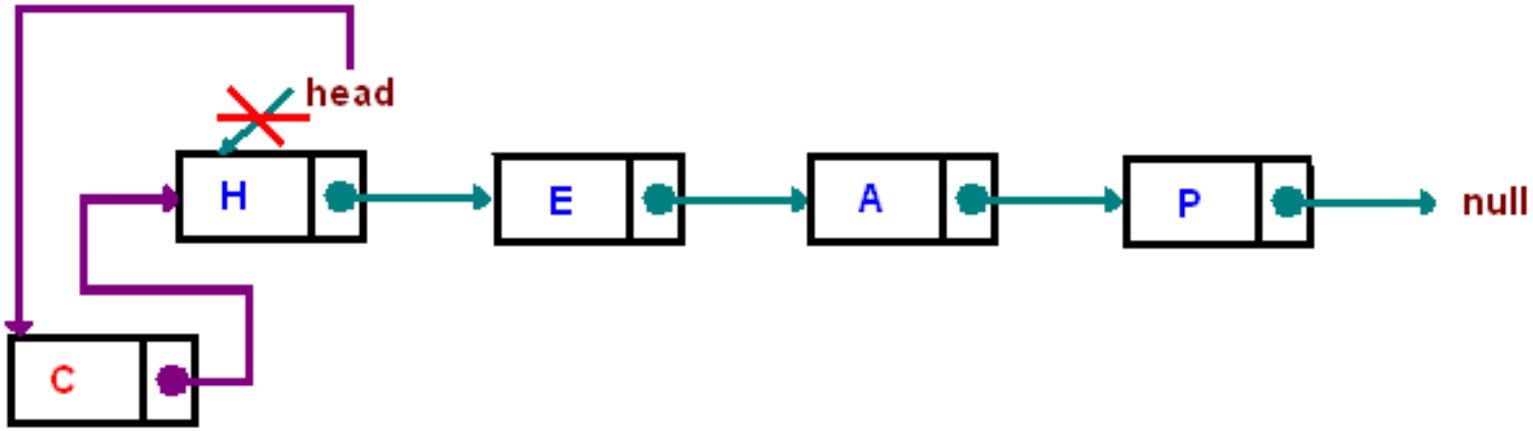


- Insert in Between
  -> InsertSorted



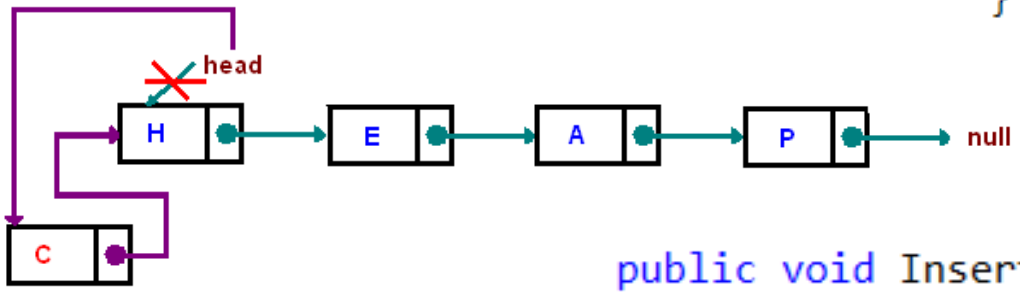
Insert Front

- Insert a new element as first element :



- public void InsertFront(Node<T> newNode)
- public void InsertFront(T value)

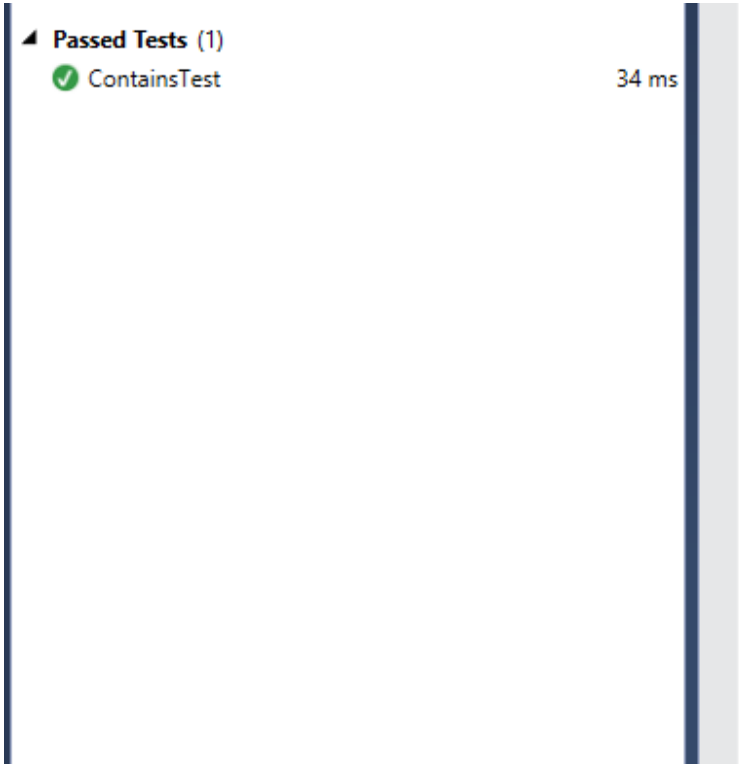
Insert Front




```
public void InsertFront(T value)
{
    InsertFront(new Node<T>(value));
}
```

```
public void InsertFront(Node<T> newNode)
{
    Node<T> temp = Head;
    if (temp == null)
        Head = newNode;
    else
    {
        newNode.Next = Head;
        Head = newNode;
    }
}
```

Test Insert Front

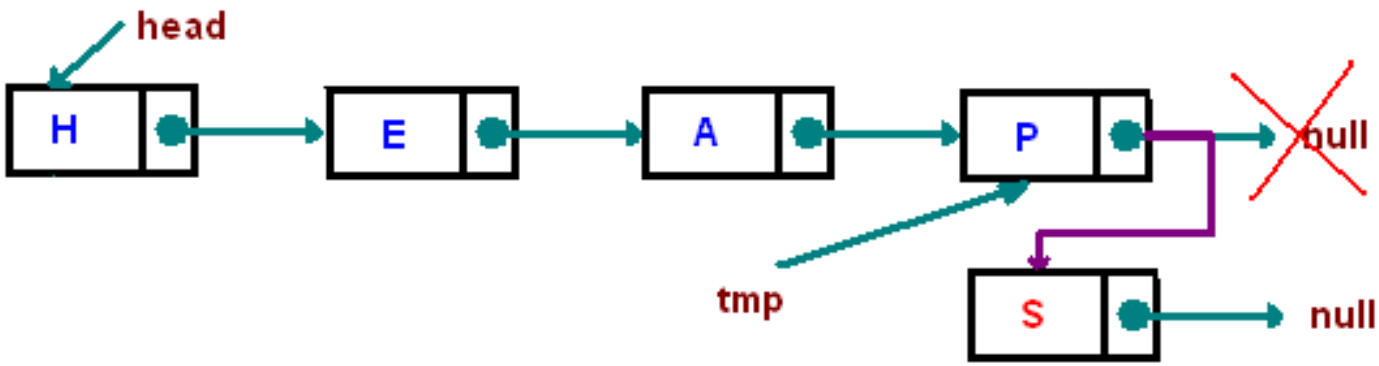


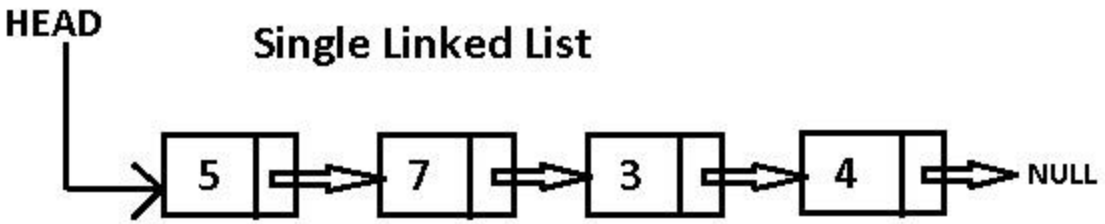

```
using System.Text;
using System.Threading.Tasks;

namespace LinkedList.Tests
{
    [TestClass()]
    public class LinkListTests
    {
        [TestMethod()]
        public void ContainsTest()
        {
            int value = 3;
            LinkList<int> l = new LinkList<int>();
            l.InsertFront(value);
            Assert.IsTrue(l.Contains(value));
        }
    }
}
```

Insert Last

- Find the last element and add an other one:

```
public void InsertLast(Node<T> newNode)
{
    Node<T> temp = Head;
    if (Head == null)
        Head = newNode;
    else
    {
        while (temp.Next != null)
            temp = temp.Next;
        temp.Next = newNode;
    }
}
```





Linked List

Node & MyLinkedList

with Head & Next

Node with Int and Next

```
class Node
{
    public int Data { get; private set; }
    public Node Next { get; set; }
    public Node(int data)
    {
        Data = data;
    }
}
```

```
class MyLinkedList
{
    //Reference to the Head - Save the Start-Node
    public Node Head { get; private set; }

    //Add a transferd node to the end of the list
    public void Append(Node n)...

    //Create a new node with the transferd value
    //Add the Node to the end of the list
    public void Append(int data)...

    //Print the data of each list-node to the console
    public void PrintList()...
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello LinkedList!");
        MyLinkedList list = new MyLinkedList();
        list.Append(3);
        list.Append(4);
        list.Append(5);
        list.Append(6);
        list.Append(7);
        list.PrintList();
```

```
Hello LinkedList!
3, 4, 5, 6, 7,
```

Append & Print

```
class MyLinkedList {
    //Reference to the Head - Save the Start-Node
    public Node Head { get; private set; }

    //Add a transferd node to the end of the list
    public void Append(Node n) {
        if (Head == null)
            Head = n;
        else {
            Node temp = Head;
            while (temp.Next != null)
                temp = temp.Next;
            temp.Next = n;
        }
    }
    //Create a new node with the transferd value
    //Add the Node to the end of the list
    public void Append(int data) {
        Node n = new Node(data);
        Append(n);
    }
```

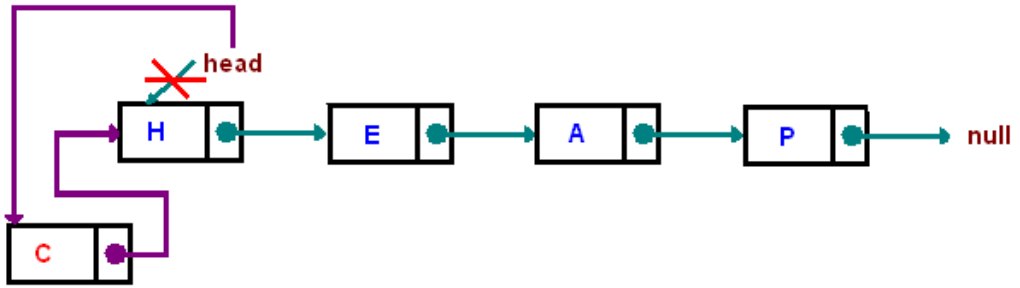
```
//Print the data of each list-node to the console
public void PrintList()
{
    if (Head == null)
    {
        Console.WriteLine("Leere Liste");
    }
    else
    {
        Node temp = Head;
        while (temp!= null)
        {
            Console.Write($"{temp.Data}, " );
            temp = temp.Next;
        }
        Console.WriteLine();
    }
}
```

InsertFront

```
//Create a new node with the transferd value
//Add the Node to the beginning of the list
public void InsertFront(int data) {
    Node n = new Node(data);
    InsertFront(n);
}
//Add a node at the beginning of the list
public void InsertFront(Node n) {
    if (n != null) {
        n.Next = Head;
        Head = n;
    }
    else
        Console.WriteLine("Ungültige Listenknoten: NULL");

}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello LinkedList!");
        MyLinkedList list = new MyLinkedList();
        list.Append(3);
        list.Append(4);
        list.Append(5);
        list.Append(6);
        list.Append(7);
        list.PrintList();

        list.InsertFront(2);
        list.InsertFront(1);
        list.PrintList();
```

```
Hello LinkedList!
3, 4, 5, 6, 7,
1, 2, 3, 4, 5, 6, 7,
```



Search & Remove a node

```
//Find & Remove a node in the middle
Node found = list.Search(3);
Node remove = list.Remove(found);
list.PrintList();

//Find the first node and remove it
found = list.Search(1);
remove = list.Remove(found);
list.PrintList();

//Find the last node and remove it
found = list.Search(7);
remove = list.Remove(found);
list.PrintList();

//Search for a invalid node/node which is not in the list
found = list.Search(10);
remove = list.Remove(found);
remove = list.Remove(new Node(10));
```

```
Hello LinkedList!
3, 4, 5, 6, 7,
1, 2, 3, 4, 5, 6, 7,
Gefunden: 3
Lösche 3
1, 2, 4, 5, 6, 7,
Gefunden: 1
Erstes Element der Liste gelöscht
2, 4, 5, 6, 7,
Gefunden: 7
Lösche 7
2, 4, 5, 6,
Nicht gefunden
Ungültiges Element zum Entfernen: NULL
```

Search

```
//Seach for a specific value in the list, return the found node or null
public Node Search(int data) {
    return Search(Head, data);
}
//Seach for a specific value in the list, return the found node or null
//Transfer a specific starting node
public Node Search(Node start, int data) {
    if (start == null) {
        Console.WriteLine("Liste leer");
        return null;
    }
    else {
        Node temp = start;
        while (temp != null) {
            if (temp.Data == data) {
                Console.WriteLine("Gefunden: " + data);
                return temp;
            }
            else
                temp = temp.Next;
        }
        Console.WriteLine("Nicht gefunden");
        return null;
    }
}
```

Remove

```
//Remove a specific node from the list, return the node or null
public Node Remove(Node r) {
    if (Head == null) {
        Console.WriteLine("Liste leer");
        return null;
    }
    else if (r == null) {
        Console.WriteLine("Ungültiges Element zum Entfernen: NULL");
        return null;
    }
    else if (Head == r) {
        Head = Head.Next;
        r.Next = null;
        Console.WriteLine("Erstes Element der Liste gelöscht");
        return r;
    }
    else {
        Node temp = Head;
        while (temp != null) {
            if (temp.Next == r) {
                temp.Next = r.Next;
                r.Next = null;
                Console.WriteLine("Lösche " + r.Data);
                return r;
            }
            else
                temp = temp.Next;
        }
    }
    return null;
}
```

IsSorted

```
//Check if a list is sorted ascending
public bool IsSorted() {
    Node temp = Head;
    bool isSorted = true;
    while (temp != null && temp.Next != null)
        if (temp.Data < temp.Next.Data)
            temp = temp.Next;
        else {
            isSorted = false;
            Console.WriteLine("Liste nicht sortiert")
            return isSorted;
        }
    Console.WriteLine("Liste ist sortiert");
    return isSorted;
}
```

```
Liste ist sortiert
Liste nicht sortiert
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello LinkedList!")
        MyLinkedList list = new MyLinkedList()
        list.Append(3);
        list.Append(4);
        list.Append(5);
        list.Append(6);
        list.Append(7);
        list.PrintList();

        //Check if a list is sorted
        list.IsSorted();
        list.Append(1);
        list.IsSorted();
```

Minimum & Maximum

```
//Find the node with the lowest value
public Node FindMinimum() {
    Node min = Head;
    Node temp = Head;
    while (temp != null)
        if (temp.Data < min.Data)
            min = temp;
        else
            temp = temp.Next;
    Console.WriteLine("Minimum: " + min.Data);
    return min;
}
//Find the node with the highest value
public Node FindMaximum() {
    Node max = Head;
    Node temp = Head;
    while (temp != null)
        if (temp.Data > max.Data)
            max = temp;
        else
            temp = temp.Next;
    Console.WriteLine("Maximum: " + max.Data);
    return max;
}
```

```
//Find and remove the node with a minimum value
Node min = list.FindMinimum();
list.Remove(min);
list.InsertFront(min.Data);
list.PrintList();
//Find the node with a maximum value
Node max = list.FindMaximum();
```

```
Minimum: 1
Lösche 1
1, 2, 4, 5, 6,
Maximum: 6
```

Insert Sorted

```
//Create a new node with the transferd value
//Insert the node, see that the list is still sorted
public void InsertSorted(int data) {
    InsertSorted(new Node(data));
}
//Insert a node, see that the list is still sorted
public void InsertSorted(Node n) {
    if (IsSorted()) {
        if (Head == null)
            Console.WriteLine("Liste leer");
        else if (n == null)
            Console.WriteLine("Ungültiger Wert für Listenknoten: NULL");
        else if (Head.Data > n.Data)
            InsertFront(n);
        else {   //Finde die korrekte Position zum Einfügen
            Node temp = Head;
            while (temp.Next != null && temp.Next.Data < n.Data)
                temp = temp.Next;
            n.Next = temp.Next;
            temp.Next = n;
        }
    }
}
```

```
list.PrintList();
list.InsertSorted(15);
list.PrintList();
list.InsertSorted(13);
list.PrintList();
list.InsertSorted(7);
list.PrintList();
```

```
1, 2, 4, 5, 6,
Maximum: 6
1, 2, 4, 5, 6,
Liste ist sortiert
1, 2, 4, 5, 6, 15,
Liste ist sortiert
1, 2, 4, 5, 6, 13, 15,
Liste ist sortiert
1, 2, 4, 5, 6, 7, 13, 15,
```

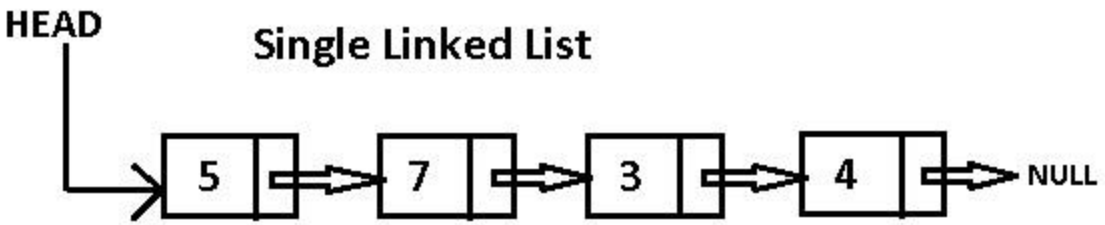
SwapNodes

```
//Swap to nodes which are already in the list.
//Don't change values, change references.
//example given with list: 6 -4 - 5 - 2- 7 - 8
//swap n1=2 and n2=6 -> list 2 - 4 - 5 - 6- 7 - 8
public void SwapNode(Node n1, Node n2) { }
```

MyLinkedList

```
class MyLinkedList
{
    //Reference to the Head - Save the Start-Node
    public Node Head { get; private set; }

    //Add a transferd node to the end of the list
    public void Append(Node n)...

    //Create a new node with the transferd value
    //Add the Node to the end of the list
    public void Append(int data)...

    //Print the data of each list-node to the console
    public void PrintList()...

    //Create a new node with the transferd value
    //Add the Node to the beginning of the list
    public void InsertFront(int data) ...
    //Add a node at the beginning of the list
    public void InsertFront(Node n)...
    //Seach for a specific value in the list, return the found node or null
    public Node Search(int data) ...
    //Seach for a specific value in the list, return the found node or null
    //Transfer a specific starting node
    public Node Search(Node start, int data) ...
    //Remove a specific node from the list, return the node or null
    public Node Remove(Node r) ...
```

```
    public bool IsSorted() ...
    //Find the node with the lowest value
    public Node FindMinimum() ...
    //Find the node with the highest value
    public Node FindMaximum()...
    //Create a new node with the transferd value
    //Insert the node, see that the list is still sorted
    public void InsertSorted(int data)...
    //Insert a node, see that the list is still sorted
    public void InsertSorted(Node n) ...

    //Swap to nodes which are already in the list.
    //Don't change values, change references.
    //example given with list: 6 -4 - 5 - 2- 7 - 8
    //swap n1=2 and n2=6 -> list 2 - 4 - 5 - 6- 7 - 8
    public void SwapNode(Node n1, Node n2) { }
```



Linked List

PersonNode & PersonLinkedList

with Head & Next

PersonNode

```
class PersonNode {
    4 Verweise
    public string Name { get; set; }
    12 Verweise
    public PersonNode Next { get; set; }
    0 Verweise
    public PersonNode() { }
    6 Verweise
    public PersonNode(string Name) {
        this.Name = Name;
    }
}
```

PersonLinkedList

```
class PersonLinkedList {
    13 Verweise
    public PersonNode Head { get; protected set; }
    4 Verweise
    public void InsertLast(PersonNode newPerson) ...
    1-Verweis
    public void PrintList() ...
    1-Verweis
    public void InsertFirst(PersonNode newNode) ...
```

InsertLast

```
class PersonLinkedList {
    public PersonNode Head { get; protected set; }
    public void InsertLast(PersonNode newPerson) {
        if (Head == null)
            Head = newPerson;
        else {
            PersonNode temp = Head;
            while (temp.Next != null)
                temp = temp.Next;
            temp.Next = newPerson;
        }
    }
```

InsertFront

```
class PersonLinkedList {
    public PersonNode Head { get; protected set; }
    public void InsertLast(PersonNode newPerson) ...
    public void PrintList() ...

    public void InsertFirst(PersonNode newNode) {
        PersonNode temp = Head;
        Head = newNode;
        newNode.Next = temp;
    }
```

PrintList

```
class PersonLinkedList {
    public PersonNode Head { get; protected set; }
    public void InsertLast(PersonNode newPerson) ...

    public void PrintList() {
        PersonNode temp = Head;
        while(temp!= null) {
            Console.WriteLine(temp.Name);
            temp = temp.Next;
        }
    }

    public void InsertFirst(PersonNode newNode) ...
```

Main Method – Test LinkedList

```
class Program {
    static void Main(string[] args) {
        PersonLinkedList list = new PersonLinkedList();
        list.InsertLast(new PersonNode("Franz"));
        list.InsertLast(new PersonNode("Kurt"));
        list.InsertLast(new PersonNode("Sepp"));
        list.InsertLast(new PersonNode("Karl"));


        list.PrintList();
```

Recursion with Linked List

Method that calls itself

PrintList Recursiv

```
public void PrintRec() {
    if (Head!= null)
        PrintRec(Head);
}
private void PrintRec(PersonNode node) {
    
}
```

PrintList Recursiv Reverse

- Print the last element first…

```
public void PrintRecReverse() {
    if (Head != null)
        PrintRecReverse(Head);
}
private void PrintRecReverse(PersonNode node) {

}
```

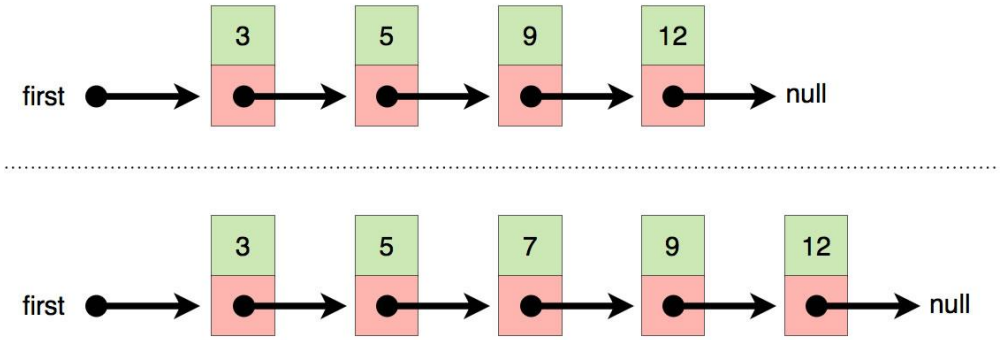
InsertLast Recursiv

```
public void InsertLastRec(PersonNode newNode) {
    if (Head != null)
        InsertLastRec(Head, newNode);
    else
        Head = newNode;
}
private void InsertLastRec(PersonNode temp,  PersonNode newNode) {

}
```

Main Method – Test Recursion

```
class Program {
    static void Main(string[] args) {
        PersonLinkedList list = new PersonLinkedList();
        list.InsertLast(new PersonNode("Franz"));
        list.InsertLast(new PersonNode("Kurt"));
        list.InsertLast(new PersonNode("Sepp"));
        list.InsertLast(new PersonNode("Karl"));
        list.InsertFirst(new PersonNode("Fabio"));
        list.InsertLastRec(new PersonNode("Stefan"));
        list.PrintList();
        Console.WriteLine("Rekursiv");
        list.PrintRec();
        Console.WriteLine("Rekursiv Reverse");
        list.PrintRecReverse();
    }
}
```

```
Fabio
Franz
Kurt
Sepp
Karl
Stefan
Rekursiv
Fabio
Franz
Kurt
Sepp
Karl
Stefan
Rekursiv Reverse
Franz
Kurt
Sepp
Karl
Stefan
Fabio
```



Sorted Linked List

PersonNode & PersonLinkedList

with Head & Next

Person List Sorted

```
class PersonListSorted
{
    public PersonNode Head { get; protected set; }
    public void InsertSorted(PersonNode node)...
    public bool IsSorted()...

    public bool Contains(string name)...
    public void PrintList()...
    public void PrintRec()...
    private void PrintRec(PersonNode person)...
    public void PrintRecReverse()...
    private void PrintRecReverse(PersonNode person)...

    public PersonNode FindNode(string name)...
    public PersonNode FindNodeRec(string name)...
    private PersonNode FindNodeRec(PersonNode person,string name)...
    public void RemoveNode(string name)...
}
```

```
class PersonNode
{
    public string Name { get; set; }
    public PersonNode Next { get; set; }
    public PersonNode() { }
    public PersonNode(string name)
    {
        this.Name = name;
    }
}
```

SortedLinkedList

```
class PersonNodeSorted {
    public PersonNode Head { get; protected set; }

    public void InsertSorted(PersonNode node) { }
    public bool IsSorted() { return true; }
    public bool Contains(string Name) { return true; }
    public void PrintList() { }
    public void PrintListRec() { }
    public void PrintListRecReverse() { }
    public PersonNode FindNode(string Name) { return null; }
    public PersonNode FindNodeRec(string Name) { return null; }
    public void RemoveNode(string name) { }

}
```

Contains

```
public class LinkList<T> : IList<T> where T: IComparable
{
    public Node<T> Head { get; protected set; }

    public bool Contains(T value)
    {
        Node<T> temp = Head;

        //while the end of the list isn't reached
        //try to find the value by stepping throw the list - comparing the data
        while (temp != null && temp.Data.CompareTo(value) != 0)
            temp = temp.Next;

        return temp != null;
    }
```

InsertSorted

```
class PersonListSorted
{
    public PersonNode Head { get; protected set; }
    public void InsertSorted(PersonNode node)
    {
        PersonNode temp = Head;
        node.Next = null;
        //Liste ist leer
        if (temp == null)
            Head = node;
        //Liste ist nicht sortiert
        else if (IsSorted() == false)
            throw new Exception("List ist nicht sortiert!");
        //Vorne einfügen
        else if (node.Name.CompareTo(temp.Name) == -1)
        {
            node.Next = temp;
            Head = node;
        }
        //Position suchen und an der richtigen Stelle einfügen
        else
        {
            while (temp.Next != null &&
                temp.Next.Name.CompareTo(node.Name) == -1)
            {
                temp = temp.Next;
            }
            node.Next = temp.Next;
            temp.Next = node;
        }
    }
```

IsSorted & Contains & Find

```
public bool IsSorted()
{
    PersonNode temp = Head;
    bool sorted = true;
    while (temp.Next != null)
    {
        if (temp.Name.CompareTo(
            temp.Next.Name) == -1)
            sorted = true;
        else
            sorted = false;
        temp = temp.Next;
    }
    return sorted;
}
```

```
public PersonNode FindNode(string name)
{
    PersonNode temp = Head;
    while (temp != null)
    {
        if (temp.Name == name)
        {
            return temp;
        }
        temp = temp.Next;
    }
    return null;
}
```

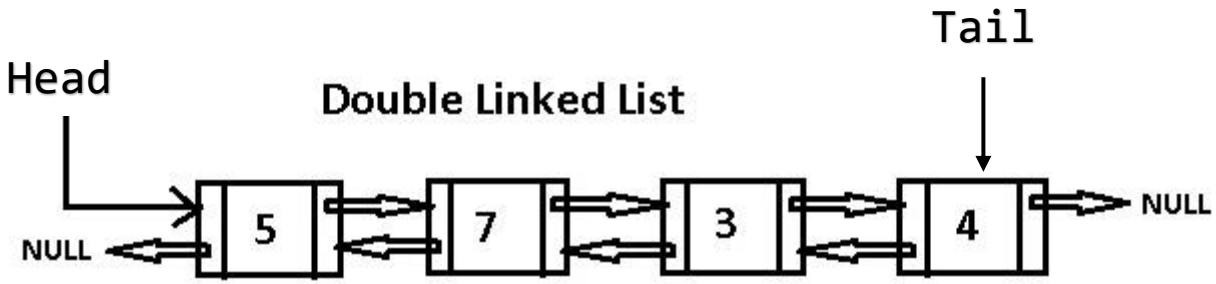
```
public bool Contains(string name)
{
    PersonNode temp = Head;
    bool contains = false;
    while (temp.Next != null)
    {
        if (temp.Name == name)
        {
            contains = true;
        }
        temp = temp.Next;
    }
    return contains;
}
```

Remove an Element

- Remove the first element
  - Special cases:
    - List ist empty
    - List has only one element
    - The first Element of the list should be removed
    - The last element of the list should be removed
    - Element in the middle of the list should be removed

Remove Node

```
public void RemoveNode(string name)
{
    PersonNode temp = Head;
    PersonNode found = null;

    //Liste leer
    if (temp == null)
        throw new Exception("Die Liste ist leer.");

    //Ersten Knoten löschen
    else if (temp.Name == name) {
        found = temp;
        Head = temp.Next;
        found.Next = null;
        Console.WriteLine("Der Knoten mit dem Wert {0} wird gelöscht.", found.Name);

    //Knoten finden und löschen
    }  else {
        while (temp.Next != null) {
            if (temp.Next.Name == name) {
                found = temp.Next;
                temp.Next = found.Next;
                found.Next = null;
                Console.WriteLine("Der Knoten mit dem Wert {0} wird gelöscht.", found.Name);
                break;
            }
            temp = temp.Next;
        }
    }

    if (found == null) {
        Console.WriteLine("Der Knoten mit dem Wert {0} wurde nicht in der Liste gefunden.",name);
    }
}
```



Double Linked List

CarNode & CarLinkedList

with Next & Prev

with Head & Tail

Vehicle Node

```
class VehicleNodeList
{
    VehicleNode Head { get; set; }
    VehicleNode Tail { get; set; }

    public VehicleNode FindNode(string brand)...
    public void InsertFront(VehicleNode vehicleNode)...
    public void InsertLast(VehicleNode vehicleNode)...

    public void Print() ...
    public void PrintReverse()...

    public void PrintRec() ...
    public void PrintRec(VehicleNode newVehicle)...
    public void PrintRecReverse()...
    public void PrintRecReverse(VehicleNode tmp)...
    public void InsertLastRec(VehicleNode newVehicle)...
    public void InsertLastRec(VehicleNode newVehicle,VehicleNode tmp)...
    public VehicleNode FindMaxHP()...
    public VehicleNode FindMinHP()...
    public VehicleNode GetNextNode(VehicleNode x)...
    public VehicleNode GetPrevNode(VehicleNode x)...

}
```

```
class VehicleNode
{
    public string Brand { get; set; }
    public int HP { get; set; }
    public VehicleNode Next { get; set; }
    public VehicleNode Prev { get; set; }

    public VehicleNode(string brand, int hp)
    {
        this.Brand = brand;
        this.HP = hp;
    }
}
```

Insert First

```
public void InsertFront(VehicleNode vehicleNode)
{

    if (Head == null)
    {
        Head = vehicleNode;
        Tail = vehicleNode;
    }
    else
    {
        vehicleNode.Next = Head;
        Head = vehicleNode;
        vehicleNode.Next.Prev = Head;
    }
}
```

Insert Last

```
public void InsertLast(VehicleNode vehicleNode)
{
    if (Head == null)
        Head = vehicleNode;
    else
    {
        VehicleNode tmp = Head;
        while (tmp.Next != null)
        {

            tmp = tmp.Next;
        }

        tmp.Next = vehicleNode;
        vehicleNode.Prev = tmp;
        Tail = vehicleNode;
    }
}
```

Print & Print Reverse

```
public void Print() ...
public void PrintReverse()
{
    VehicleNode tmp = Tail;
    while (tmp != null)
    {
        Console.WriteLine(tmp.Brand + ", " + tmp.HP + "HP");
        tmp = tmp.Prev;
    }
}
public void PrintRec() ...
private void PrintRec(VehicleNode newVehicle)...
public void PrintRecReverse()...
private void PrintRecReverse(VehicleNode tmp)...
```

Find brand - Find Min & Max Value

```
public VehicleNode FindNode(string brand){
    VehicleNode tmp = Head;
    while (tmp!=null) {
        if (tmp.Brand == brand )
            return tmp;
        tmp = tmp.Next;
    }
    return null;
}
```

```
public VehicleNode FindMaxHP()
{
    VehicleNode tmp = Head;
    VehicleNode maxHPVehicle = Head;
    while (tmp != null)
    {
        if (tmp.HP >maxHPVehicle.HP)
        {
            maxHPVehicle = tmp;
        }
        tmp = tmp.Next;
    }
    return maxHPVehicle;
}
```

```
public VehicleNode FindMinHP()
{
    VehicleNode tmp = Head;
    VehicleNode minHPVehicle = Hea
    while (tmp != null)
    {
        if (tmp.HP < minHPVehicle.
        {
            minHPVehicle = tmp;
        }
```

Get Previous & Next

```
public VehicleNode GetNextNode(VehicleNode x)
{
    return x.Next;
}
public VehicleNode GetPrevNode(VehicleNode x)
{
    return x.Prev;
}
```

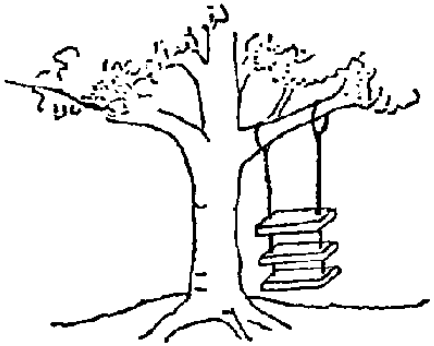
Test the Main

```
static void Main(string[] args) {
    VehicleNodeList list = new VehicleNodeList();
    list.InsertLast(new VehicleNode("Audi", 120));
    list.InsertLast(new VehicleNode("Bmw", 155));
    list.InsertLast(new VehicleNode("Seat", 75));
    list.InsertLast(new VehicleNode("Ford", 90));
    list.InsertFront(new VehicleNode("Skoda", 105));
    list.InsertFront(new VehicleNode("Mini", 110));
    list.InsertFront(new VehicleNode("Porsche", 250));
    list.InsertLastRec(new VehicleNode("Peugeot", 55));
    list.InsertLastRec(new VehicleNode("Jaguar", 255));
    list.InsertLastRec(new VehicleNode("Fiat", 121));
    Console.WriteLine("\tAusgabe\n");
    list.Print();
    Console.WriteLine("\tAusgabe Rückwärts\n");
    list.PrintReverse();
    Console.WriteLine("\tRekursive Ausgabe\n");
    list.PrintRec();
    Console.WriteLine("\tRekursive Ausgabe Rückwärts\n");
    list.PrintRecReverse();
    Console.WriteLine("\tMinimale Pferdestärke");
    VehicleNode min = list.FindMinHP();
    Console.WriteLine(min.Brand);
    Console.WriteLine("\tMaximale Pferdestärke");
    VehicleNode max = list.FindMaxHP();
    Console.WriteLine(max.Brand);
    Console.WriteLine("\tFinde den BMW:");
    VehicleNode find = list.FindNode("Bmw");
    Console.WriteLine(find.Brand + "\n");
    Console.WriteLine("Next:\t" + list.GetNextNode(max).Brand);
    Console.WriteLine("Previous: \t" + list.GetPrevNode(min).Brand);
}
```

```
        Ausgabe

Porsche 250HP, Mini 110HP, Skoda 105HP, Audi 120H
        Ausgabe Rückwärts

Fiat 121HP, Jaguar 255HP, Peugeot 55HP, Ford 90HP
        Rekursive Ausgabe

Porsche 250HP, Mini 110HP, Skoda 105HP, Audi 120H
        Rekursive Ausgabe Rückwärts

Fiat 121HP, Jaguar 255HP, Peugeot 55HP, Ford 90HP
        Minimale Pferdestärke
Peugeot
        Maximale Pferdestärke
Jaguar
        Finde den BMW:
Bmw

Next:   Fiat
Previous:       Ford
Drücken Sie eine beliebige Taste . . .
```

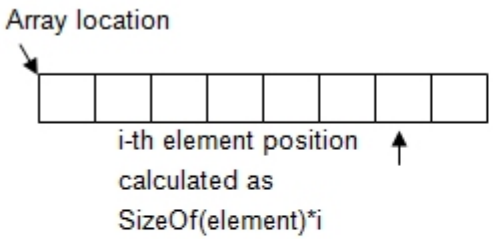
C# Generics

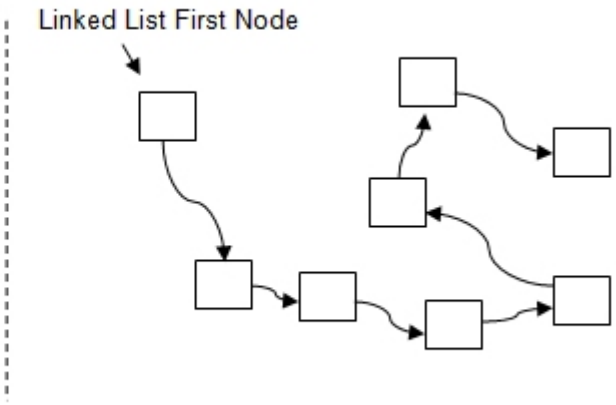
SEW



Content

- Definition of Generics
- Stack as a Generic Type
- Generic Benefit
- Multiple Generic Types
- Generic Linked List
- Where-Clause





type-safe classes without compromising type safety, performance, or productivity.

Generics

genui, genitus, ‚zeugen', ‚hervorbringen', ‚verursachen') ist die Eigenschaft eines materiellen oder abstrakten Objekts, insbesondere eines Begriffs, nicht auf Spezifisches, also auf unterscheidende Eigenheiten Bezug zu nehmen, sondern im Gegenteil sich auf eine ganze Klasse, Gattung oder Menge anwenden zu lassen bzw. ...

Generisch – Wikipedia
https://de.wikipedia.org/wiki/Generisch

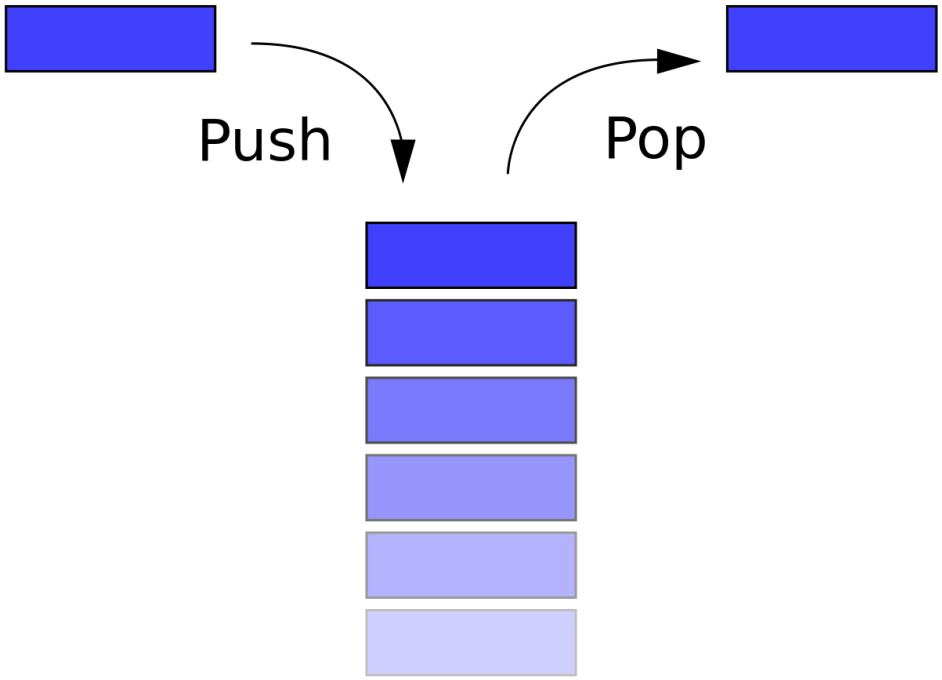
Generic Class

- Use the < and > brackets, enclosing a generic type parameter.

```
public class GenericClass<T>
{
    public T msg;
    public void genericMethod(T name, T location)
    {
        Console.WriteLine("{0}", msg);
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("Location: {0}", location);
    }
}
```

Generic Method

```
public class GenericClass<T> where T : class
{
    public T msg;
    public void genericMethod<X>(T name, T location) where X : class
    {
        Console.WriteLine("{0}", msg);
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("Location: {0}", location);
    }
}
```



Generic Stack

Implement a Generic Stack

Initialize the stack with a maximum amount

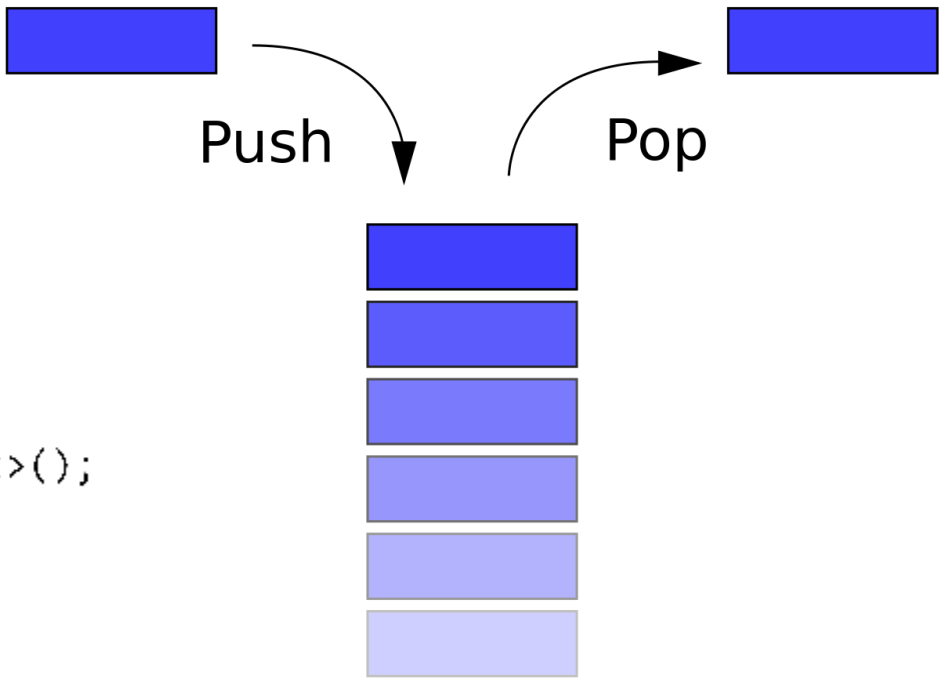
Push puts an element into the stack at the top

Pop gets an element of the stack from the top

Generic Stack

- For example, here is how you define and use a generic stack:

```
public class Stack<T>
{
    T[] m_Items;
    public void Push(T item)
    {...}
    public T Pop()
    {...}
}
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```



Stack Implementation

```
class Stack<T>
{
    private readonly int size;
    private T[] elements;
    private int pointer = 0;

    public Stack(int size)...

    public void Push(T element)...

    public T Pop()...

    public int Length...
}
```

Stack<T>

Konstruktor

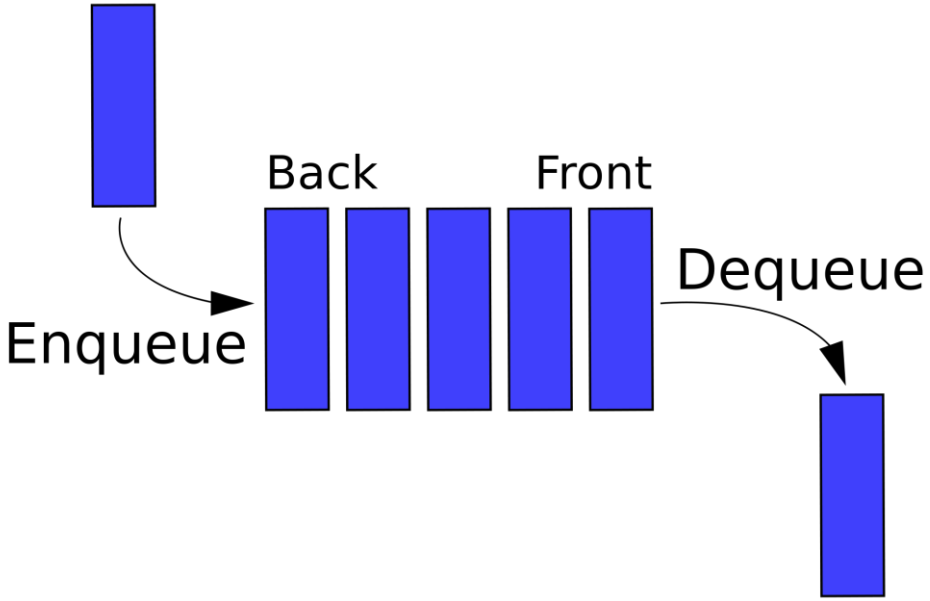
Push & Pop

Length

```
class Stack<T>
{
    private readonly int size;
    private T[] elements;
    private int pointer = 0;

    public Stack(int size)...

    public void Push(T element) {
        if (pointer >= this.size)
            throw new StackOverflowException();
        elements[pointer] = element;
        pointer++;
    }

    public T Pop() {
        pointer--;
        if (pointer >= 0)
            return elements[pointer];
        else {
            pointer = 0;
            throw new InvalidOperationException("Der Stack ist leer");
        }
    }

    public int Length {
        get { return this.pointer; }
    }
}
```

Exercise – Generic Queue

- Implement a generic queue
  - with an array
  - Dequeue – removes the first element
  - Enqueue – adds an element



```
//Class QueueSorted
public class QueueSorted<T> where T : IComparable, new() {
    private List<T> list = new List<T>();

    //Add an Item - find the right place to insert it
    //CompareTo liefert < 0 retour wenn kleiner
    //                  = 0 retour wenn gleich
    //                  > 0 retour wenn größer
    public void InsertSorted(T item) {
        int i = 0;
        while (i < list.Count && list.ElementAt(i).CompareTo(item) < 0)
                i++;
        list.Insert(i, item);
    }

    public void PrintQueue() {
        foreach (var item in list) {
            Console.WriteLine(item.ToString());
        }
    }

}
```

Generic Benefit

- reuse code - no code doubling
  - types and internal data can change without causing code bloat, regardless of whether you are using value or reference types
- test it once
  - develop, test, and deploy code once, reuse it with any type, including future types, all with full compiler support and type safety.

Multiple Generic Types

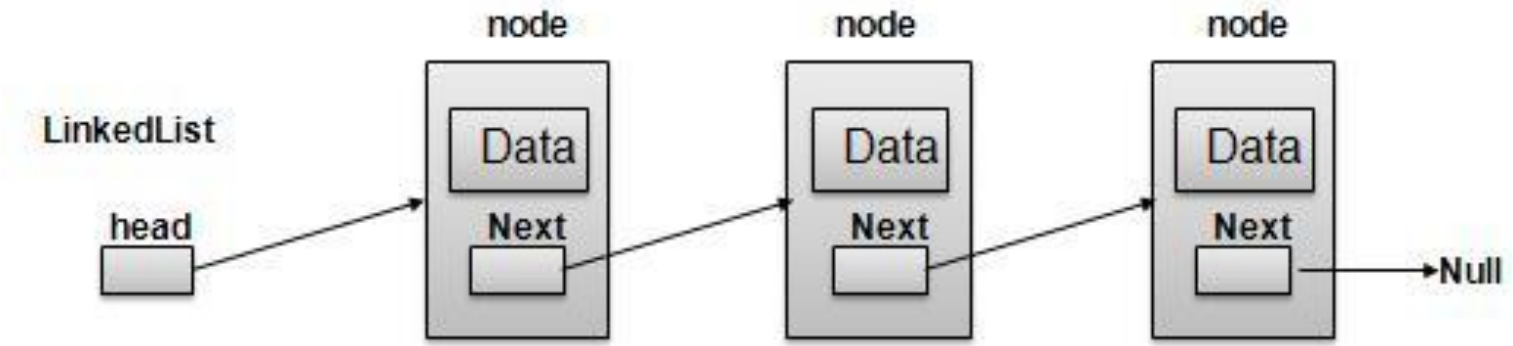
- A single type can define multiple generic-type parameters.

- `class NodeMultipleDouble<T, U> { }`
- `class NodeMultipleTriple<K, V, U> { }`

```
public class GenericClass<T, X> where T : class where X : struct
{
    // Your Implementation
}
```

Example Generic Linked List

- A class Node has some data
  - a key and
  - an item
- To navigate threw the list
  - use a reference to the next node

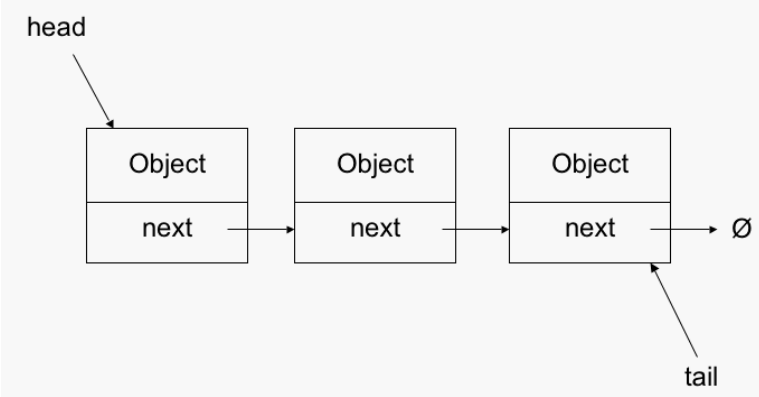


Class Node

```
class Node<K,T>
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
   public Node()
   {
      Key      = default(K);
      Item     = defualt(T);
      NextNode = null;
   }
   public Node(K key,T item,Node<K,T> nextNode)
   {
      Key      = key;
      Item     = item;
      NextNode = nextNode;
   }
}
```

Node<K,T>
K key
T item
Node<K,T> next

Class Linked List




```
public class LinkedList<K,T>
{
    Node<K,T> m_Head;
    public LinkedList()
    {
        m_Head = new Node<K,T>();
    }
    public void AddHead(K key,T item)
    {
        Node<K,T> newNode = new Node<K,T>(key,item,m_Head.NextNode);
        m_Head.NextNode = newNode;
    }
}
```

Where-Clause

- Multiple Type Parameter & Where Clause

- The Key of an Dictionary should be compareable
- The Item could also follow a definition

```
class Dictionary<TKey, TVal>
    where TKey : IComparable, IEnumerable
    where TVal : IMyInterface
{
```

Where-Clause

- any operations, fields, methods, properties, etc that you attempt to use of type **T** must be available at the lowest common denominator type: **object**

- where clause is used to specify constraints on the types that can be used as arguments for a type parameter defined in a generic declaration

- For example:

```
public class MyGenericClass<T> where T:IComparable { }
```

Different type of constraints:

Constraint	Description
where T : struct	The type argument must be a value type.
where T : unmanaged	The type of argument must not be a reference type.
where T : class	The type argument must be a reference type.
where T : new()	The type argument must have a public parameterless constructor.
where T : <base class name>	The type argument must be or derive from the specified base class.
where T : <interface name>	The type argument must be or implement the specified interface.
where T : U	The type argument supplied for T must be or derive from the argument supplied for U.

Generic Class where T : class

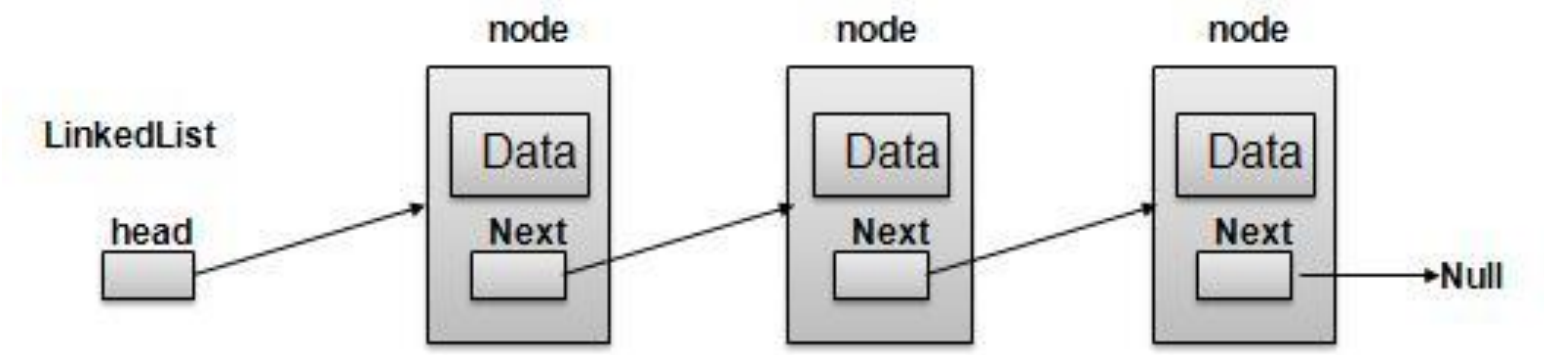
```
public class GenericClass<T> where T: class
{
    public T msg;
    public void genericMethod(T name, T location)
    {
        Console.WriteLine("{0}", msg);
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("Location: {0}", location);
    }
}
```

```
// Instantiate Generic Class with Constraint
GenericClass<string> gclass = new GenericClass<string>();
GenericClass<User> gclass1 = new GenericClass<User>();
// Compile Time Error
//GenericClass<int> gclass11 = new GenericClass<int>();
```

Summary Generics

- In c#, constraints are used to restrict a generics to accept only the particular type of placeholders.
- By using **where** keyword, we can apply a constraints on generics.
- In c#, we can apply a multiple constraints on generic class or methods based on our requirements.
- In c#, we have a different type of constraints available, those are class, structure, unmanaged, new(), etc.

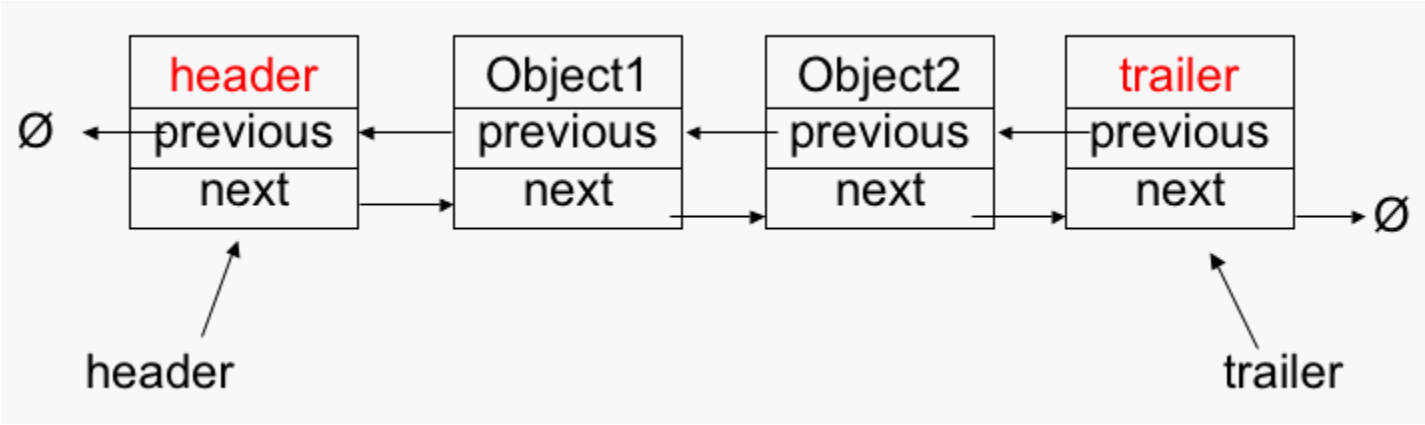
Generic Linked Lists



Class Node<T>

```
public class Node<T> where T : IComparable
{
    public T Data { get; set; }
    public Node<T> Next { get; set; }
    public Node<T> Previous { get; set; }
```

Node<T>
T data
Node<T> next
Node<T> prev



Class Node<T>

```
public class Node<T> where T : IComparable
{
    public T Data { get; set; }
    public Node<T> Next { get; set; }
    public Node<T> Previous { get; set; }

    public Node(){}
    public Node(T value)
    {
        this.Data = value;
        Next = null;
        Previous = null;
    }
}
```

Node<T>
T data
Node<T> next
Node<T> prev

ExpandedNode<K,T>

- Add additional data to the node

```
class ExpandedNode<K, T> : Node<K> where K : IComparable
{
    public T ExpandedData { get; set; }

    public ExpandedNode(K data, T expandedData)
    {
        base.Data = data;
        this.ExpandedData = expandedData;
        base.Next = null;
        base.Previous = null;
    }
}
```

Node<K,T>
K data
T expandedData
Node<T> next
Node<T> prev

Klasse Node

```
public abstract class Node<K,T> where K : IComparable<K>
{
    33 Verweise
    public K Key { get; set; }
    10 Verweise
    public T Value { get; set; }

    2 Verweise
    public Node(K key, T value)
    {
        this.Key = key;
        this.Value = value;
    }

    5 Verweise
    public override string ToString()
    {
        return Key.ToString() + ": " + Value.ToString();
    }
}
```

LinkList<T> and Contains

```
public class LinkList<T> : IList<T> where T: IComparable
{
    public Node<T> Head { get; protected set; }

    public bool Contains(T value)
    {
        Node<T> temp = Head;

        //while the end of the list isn't reached
        //try to find the value by stepping throw the list - comparing the data
        while (temp != null && temp.Data.CompareTo(value) != 0)
            temp = temp.Next;

        return temp != null;
    }
```

IList<T>

Possible

List-Methods for Node<T>

Node<T>
T data
Node<T> next

```
public interface IList<T> where T: IComparable
{
    //Add a Node
    void InsertFront(Node<T> newNode);
    void InsertLast(Node<T> newNode);
    void InsertSorted(Node<T> newNode);

    //Add a Value
    void InsertFront(T value);
    void InsertLast(T value);
    void InsertSorted(T value);

    //Print
    void PrintList();

    //Search and ...
    bool Contains(T value);
    Node<T> Find(T value);

    //Remove: remove the node from the list and return it
    Node<T> Remove(T value);
```

Linked List – Part 1

```
public class LinkedList<K, T> : AList<K, T> where K : IComparable<K>
{
    private ISortBehaviour<K, T> sortBeh;

    8 Verweise
    public LinkedList() : base()  {  }

    0 Verweise
    public LinkedList(ListNode<K, T>[] arr)...

    3 Verweise
    protected override void InsertFromArray(ListNode<K, T>[] arr)...

    0 Verweise
    public void SetSortBehaviour(ISortBehaviour<K, T> beh)...
    0 Verweise
    public void Sort()
    {
        sortBeh.Sort(this);
    }
```

Linked List – Part 2

```
public void InsertFirst(ListNode<K, T> node)...
0 Verweise
public void InsertFirst(K key, T value)...

10 Verweise
public void InsertLast(ListNode<K, T> node)...
8 Verweise
public void InsertLast(K key, T value)...

4 Verweise
public void Swap(ListNode<K, T> n, ListNode<K, T> m)...
0 Verweise
public void Swap(K key1, K key2)...
```

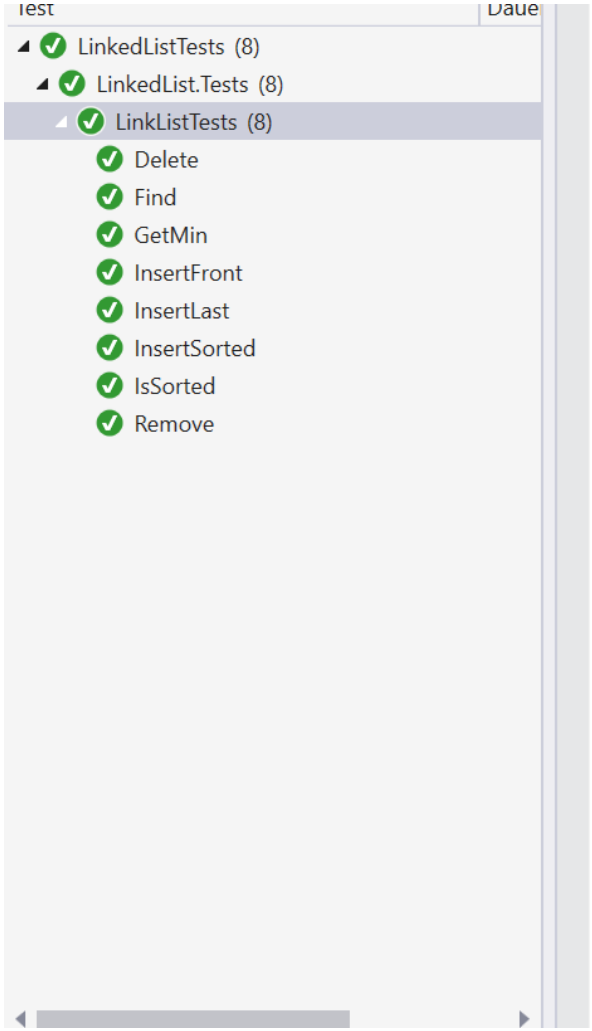
Print the List

```
public void PrintList()
{
    Console.WriteLine(this.ToString());
}

public override String ToString()
{
    StringBuilder sb = new StringBuilder();
    Node<T> temp = Head;
    while (temp != null)
    {
        sb.Append(String.Format("{0}, ", temp.Data));
        temp = temp.Next;
    }
    return sb.ToString();
}
```

Insert Sorted

```
public void InsertSorted(ListNode<K, T> node)
{
    if (Head == null) {
        Head = node;
        Tail = node;
        return;
    }
    if (node.Key.CompareTo(Head.Key) <= 0) {
        InsertHead(node);
        return;
    }
    ListNode<K, T> cur = Head;
    ListNode<K, T> tmp = cur.Next;
    while (node.Key.CompareTo(cur.Key) > 0) {
        if (tmp == null) {
            InsertTail(node);
            return;
        } else {
            cur = tmp;
            tmp = cur.Next;
        }
    }
    tmp = cur;
    cur = tmp.Prev;

    cur.Next = node;
    node.Prev = cur;
    node.Next = tmp;
    tmp.Prev = node;
}
0 Verweise
public void InsertSorted(K key, T value)
{
    InsertSorted(new ListNode<K, T>(key,value));
}
```

Implement Unit-Tests



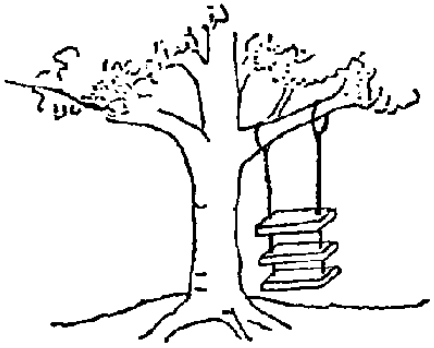

```
31          [TestMethod()]
32          public void InsertLast()
33          {
34              int value = 10;
35              LinkList<int> l = new LinkList<int>();
36              int[] arr = { 5, 9, 3, 19, 45, 66, 22, 56 };
37              int count = arr.Length + 4;
38              l.InsertLast(value);
39              l.AddArray(arr);
40              l.InsertLast(value+20);
41              l.InsertLast(value+30);
42              l.InsertLast(value+40);
43              l.PrintList();
44              Assert.AreEqual(count,l.Count());
45          }
46          [TestMethod()]
47          public void Find()
48          {
49              LinkList<int> l = new LinkList<int>();
50
51              int[] arr = { 5, 9, 3, 19, 45, 66, 22, 56 };
52              l.AddArray(arr);
53              Node<int> foundval = l.Find(45);
54              Assert.AreEqual(foundval.Data, 45);
55          }
56          [TestMethod()]
57          public void Delete()
58          {
59              int value = 25;
60              LinkList<int> l = new LinkList<int>();
61              l.InsertLast(25);
62              l.InsertLast(50);
```

Sorting with Strategy Pattern

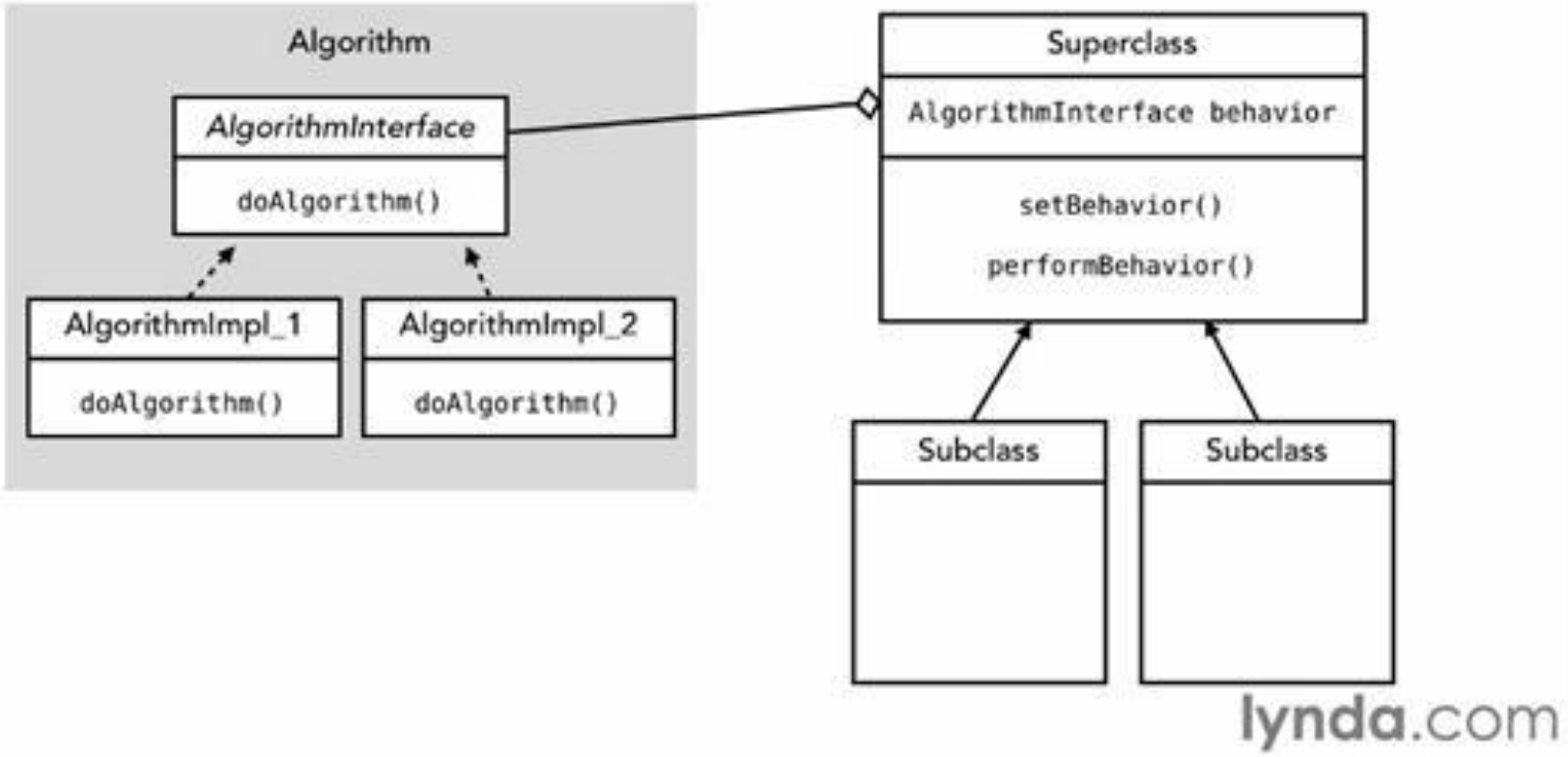
- Insertion Sort
- Selection Sort
- Bubble Sort
- Quick Sort
- Merge Sort

Strategy Pattern

We define multiple algorithms and let client application pass the algorithm to be used as a parameter



The Strategy Pattern



Using ISortBehaviour

```
8 Verweise
public interface ISortBehaviour<K, T> where K : IComparable<K>
{
    7 Verweise
    void Sort(LinkedList<K, T> list);
}
```

```
public class LinkedList<K, T> : AList<K, T> where K : IComparable<K>
{
    private ISortBehaviour<K, T> sortBeh;

    0 Verweise
    public void SetSortBehaviour(ISortBehaviour<K, T> beh)
    {
        this.sortBeh = beh;
    }
```

Insertion Sort

```
public class InsertionSort<K, T> : ISortBehaviour<K, T>
    where K : IComparable<K>
{
    7 Verweise
    public void Sort(LinkedList<K, T> list)
    {
        SortedList<K, T> sortedList = new SortedList<K, T>();

        while (list.Head != null)
            sortedList.InsertSorted(list.Remove(list.Head));

        list.Override(sortedList);
    }
}
```

```
public void Override(AList<K, T> list)
{
    this.Head = list.Head;
    this.Tail = list.Tail;
}
```

Selection Sort

```
public class SelectionSort<K, T> : ISortBehaviour<K, T> where K : IComparable<K>
{
    7 Verweise
    public void Sort(LinkedList<K, T> list)
    {
        LinkedList<K, T> sortedList = new LinkedList<K, T>();

        while (list.Head != null)
            sortedList.InsertLast(list.Remove(GetMinimum(list)));

        list.Override(sortedList);
    }
    1-Verweis
    ListNode<K, T> GetMinimum(LinkedList<K, T> list)
    {
        ListNode<K, T> minimum = list.Head;
        for(ListNode<K, T> curr = list.Head; curr != null; curr = curr.Next)
            if (curr.Key.CompareTo(minimum.Key) < 0)
                minimum = curr;
        return minimum;
    }
}
```

Bubble Sort

```
public class BubbleSort<K, T> : ISortBehaviour<K, T> where K : IComparable<K>
{
    7 Verweise
    public void Sort(LinkedList<K, T> list)
    {
        if (list.Head == null)
            return;

        bool sorted = false;
        while (!sorted)
        {
            sorted = true;

            for (ListNode<K, T> curr = list.Head; curr.Next != null;)
            {
                if (curr.Key.CompareTo(curr.Next.Key) > 0)
                {
                    list.Swap(curr, curr.Next);
                    sorted = false;
                }
                else
                    curr = curr.Next;
            }
        }
    }
}
```

Swap Nodes

```
public void Swap(ListNode<K, T> n, ListNode<K, T> m)
{
    if (n == Head)
        Head = m;
    else if (m == Head)
        Head = n;
    if (n == Tail)
        Tail = m;
    else if (m == Tail)
        Tail = n;

    if (n.Next == m)
    {
        ListNode<K, T> np = n.Prev, mn = m.Next;

        if (np != null)
            np.Next = m;
        m.Prev = np;
        m.Next = n;
        n.Prev = m;
        if (mn != null)
            mn.Prev = n;
        n.Next = mn;
    }
    else if (m.Next == n)
    {
        ListNode<K, T> mp = m.Prev, nn = n.Next;
        if (mp != null)
            mp.Next = n;
        n.Prev = mp;
        m.Prev = n;
        n.Next = m;
        if (nn != null)
            nn.Prev = m;
        m.Next = nn;
    }
    else
    {
        ListNode<K, T> np = n.Prev, nn = n.Next;
        ListNode<K, T> mp = m.Prev, mn = m.Next;

        if (np != null)
            np.Next = m;
        m.Prev = np;
        if (nn != null)
            nn.Prev = m;
        m.Next = nn;

        if (mp != null)
            mp.Next = n;
        n.Prev = mp;
        if (mn != null)
            mn.Prev = n;
        n.Next = mn;
    }
```

```
public void Swap(K key1, K key2)
{
    ListNode<K, T> cur = Head;
    ListNode<K, T> n = null;
    ListNode<K, T> m = null;
    while (cur != null)
    {
        if (cur.Key.CompareTo(key1) == 0)
            n = cur;
        if (cur.Key.CompareTo(key2) == 0)
            m = cur;
        cur = cur.Next;
    }
    Swap(n,m);
}
```

```
public class QuickSort<K, T> : ISortBehaviour<K, T> where K : IComparable<K>
{
    int count;
    7 Verweise
    public void Sort(LinkedList<K, T> list)
    {
        for (ListNode<K, T> curr = list.Head; curr != null; curr = curr.Next)
            count++;

        QuickSortLeft(list, 0, count - 1);
    }

    3 Verweise
    public void QuickSortLeft(LinkedList<K, T> list, int left, int right)
    {
        int i;

        if (left < right)
        {
            i = SplitRight(list, left, right);
            QuickSortLeft(list, left, i - 1);
            QuickSortLeft(list, i + 1, right);
        }
    }
```

Quick Sort SplitLeft

QuickSort Split Right

```
public int SplitRight(LinkedList<K, T> list, int left, int right)
{
    int k;

    ListNode<K,T> pivot = GetNode(right, list);
    int i = left -1;

    for (k = left; k <= right-1; k++)
    {
        if (pivot.Key.CompareTo(GetNode(k,list).Key) >= 0)
        {
            i++;
            list.Swap(GetNode(i, list),
                GetNode(k, list));

        }
    }
    i++;
    list.Swap(pivot,
        GetNode(i,list));
    return i;
}
```

```
public ListNode<K, T> GetNode(int index, LinkedList<K, T> list)
{
    ListNode<K, T> curr = list.Head;
    for (int i = 0; i < index; i++)
    {
        curr = curr.Next;
    }
    return curr;
}
```

Merge Sort

```
public class MergeSort<K, T> : ISortBehaviour<K, T> where K : IComparable<K>
{
    int count;

    7 Verweise
    public void Sort(LinkedList<K, T> list)
    {
        ListNode<K, T> curr = list.Head;
        while (curr != null)
        {
            count++;
            curr = curr.Next;
        }
        MergeSorter(list, 0, count - 1);
    }

    3 Verweise
    private void MergeSorter(LinkedList<K, T> list, int left, int right)
    {
        int mid;

        if (left < right)
        {
            mid = (left + right) / 2;
            MergeSorter(list, left, mid);
            MergeSorter(list, mid + 1, right);
            Merge(list, left, right, mid);
        }
    }
```

Merge

```
private void Merge(LinkedList<K, T> list, int left, int right, int mid)
{
    int i, j;
    LinkedList<K, T> newlist = new LinkedList<K, T>();

    for (int n = 0; n < left; n++)
        newlist.InsertLast(Clone(GetNode(n, list)));

    i = left;
    j = mid + 1;

    while (i <= mid && j <= right)
    {
        if (GetNode(i, list).Key.CompareTo(GetNode(j, list).Key) < 0)
            newlist.InsertLast(Clone(GetNode(i++, list)));
        else
            newlist.InsertLast(Clone(GetNode(j++, list)));
    }

    while (i <= mid)
        newlist.InsertLast(Clone(GetNode(i++, list)));
    while (j <= right)
        newlist.InsertLast(Clone(GetNode(j++, list)));

    for (int n = right + 1; n < count; n++)
        newlist.InsertLast(Clone(GetNode(n, list)));

    list.Override(newlist);
}
```

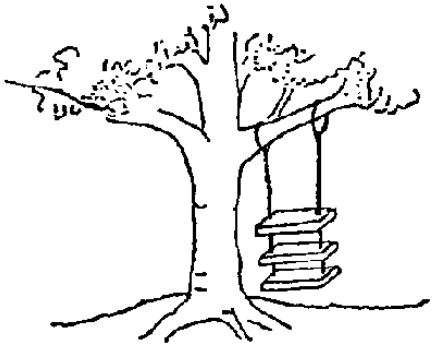
```
private ListNode<K,T> GetNode(int index, LinkedList<K, T> list)
{
    ListNode<K,T> curr = list.Head;
    for (int i = 0; i < index; i++)
    {
        curr = curr.Next;
    }
    return curr;
}
```

6 Verweise

```
private ListNode<K,T> Clone(ListNode<K, T> n)
{
    return new ListNode<K, T>(n.Key, n.Value);
}
```

Iterator Pattern

Iterator pattern falls under behavioral pattern category.



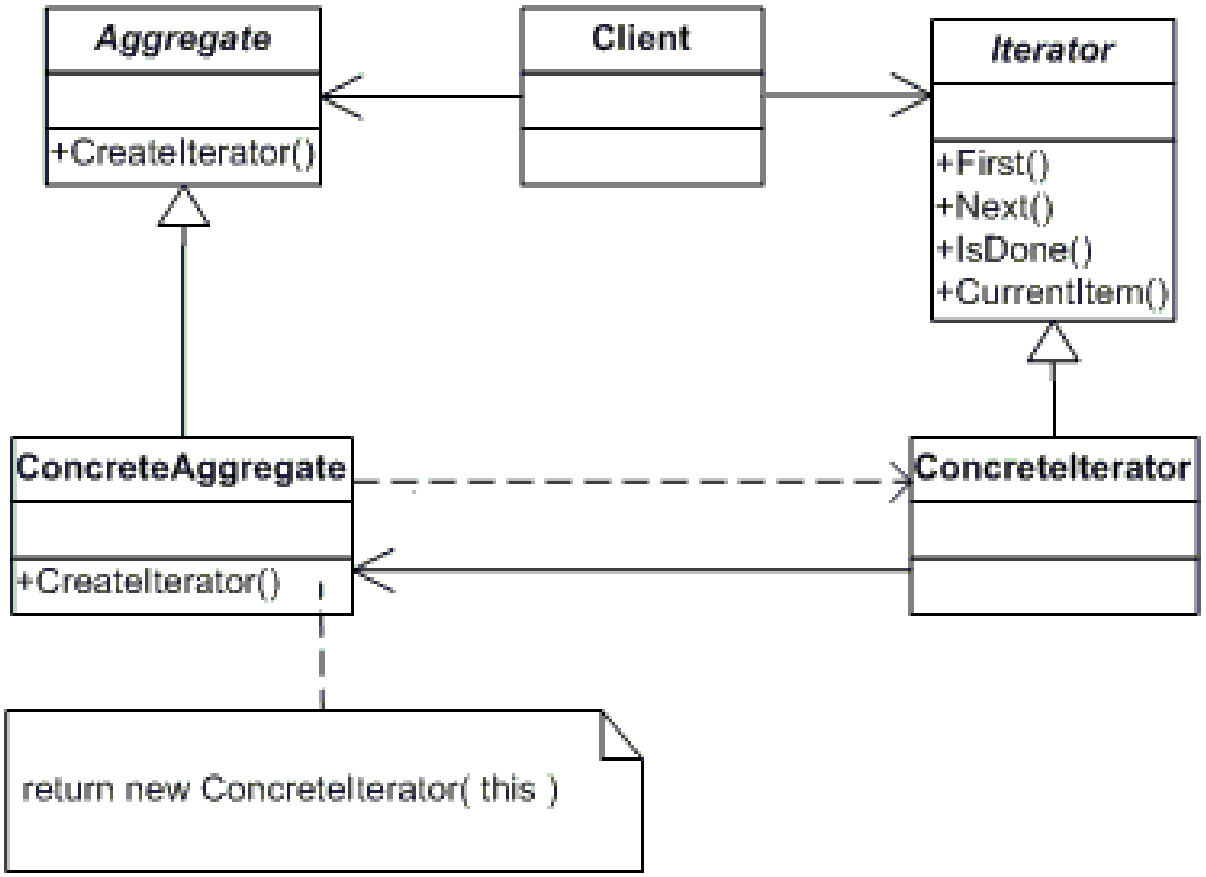
Iterator Pattern

This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Promote to "full object status" the traversal of a collection.
- Polymorphic traversal

UML Diagramm



Participants

- Iterator (AbstractIterator)
  - defines an interface for accessing and traversing elements.
- ConcreteIterator (Iterator)
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- Aggregate (AbstractCollection)
  - defines an interface for creating an Iterator object
- ConcreteAggregate (Collection)
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Implementation

```
/// <summary>
/// The 'Aggregate' abstract class
/// </summary>
abstract class Aggregate
{
    public abstract Iterator CreateIterator();
}

/// <summary>
/// The 'ConcreteAggregate' class
/// </summary>
class ConcreteAggregate : Aggregate
{
    private ArrayList _items = new ArrayList();

    public override Iterator CreateIterator()
    {
        return new ConcreteIterator(this);
    }
```

Aggregate

```
    // Gets item count
    public int Count
    {
        get { return _items.Count; }
    }

    // Indexer
    public object this[int index]
    {
        get { return _items[index]; }
        set { _items.Insert(index, value); }
    }
}
```

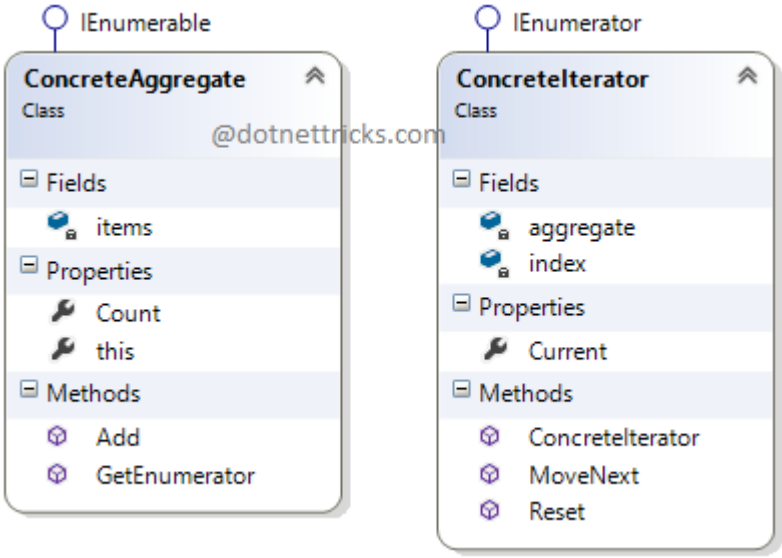
Iterator

```
/// <summary>
/// The 'Iterator' abstract class
/// </summary>
abstract class Iterator
{
    public abstract object First();
    public abstract object Next();
    public abstract bool IsDone();
    public abstract object CurrentItem();
}

/// <summary>
/// The 'ConcreteIterator' class
/// </summary>
class ConcreteIterator : Iterator
{
    private ConcreteAggregate _aggregate;
    private int _current = 0;

    // Constructor
    public ConcreteIterator(ConcreteAggregate aggregate)
    {
        this._aggregate = aggregate;
    }

    // Gets first iteration item
    public override object First()
    {
        return _aggregate[0];
    }
```

```
    // Gets next iteration item
    public override object Next()
    {
        object ret = null;
        if (_current < _aggregate.Count - 1)
        {
            ret = _aggregate[++_current];
        }

        return ret;
    }

    // Gets current iteration item
    public override object CurrentItem()
    {
        return _aggregate[_current];
    }

    // Gets whether iterations are complete
    public override bool IsDone()
    {
        return _current >= _aggregate.Count;
    }
```

Main: Using an Iterator

```
static void Main()
{
    ConcreteAggregate a = new ConcreteAggregate();
    a[0] = "Item A";
    a[1] = "Item B";
    a[2] = "Item C";
    a[3] = "Item D";

    // Create Iterator and provide aggregate
    Iterator i = a.CreateIterator();

    Console.WriteLine("Iterating over collection:");

    object item = i.First();
    while (item != null)
    {
        Console.WriteLine(item);
        item = i.Next();
    }
```



Iterator for LinkList<T>

Using IEnumerable & IEnumerator Interface of C# Library System.Collections;

LinkListIterator

```
public interface IList<T>:IEnumerable where T: IComparable
{
    //Add a Node
    void InsertFront(Node<T> newNode);
    void InsertLast(Node<T> newNode);
    void InsertSorted(Node<T> newNode);
```

```
public class LinkList<T> : IList<T> where T : IComparable
{
    public Node<T> Head { get;  set; }
    public Node<T> Tail { get; set; }

    public IEnumerator GetEnumerator() {
        return new LinkListIterator<T>(this);
    }
```

```
class LinkListIterator<T> : IEnumerator where T : IComparable {
    public LinkList<T> list;
    private Node<T> tmp;

    public object Current {
        get {
            return tmp;
        }
    }

    public LinkListIterator(LinkList<T> list) {
        this.list = list;
    }

    public bool MoveNext() {
        if (tmp == null) {
            tmp = list.Head;
            return true;
        }
        else if (tmp.Next == null) {
            return false;
        }
        else {
            tmp = tmp.Next;
            return true;
        }
    }

    public void Reset() {
        tmp = null;
    }

    public void Dispose() {
        //nothing to do
    }
}
```