Структури от данни

Данни на една програма

- Памет;
- Тип;
- Структури от данни;

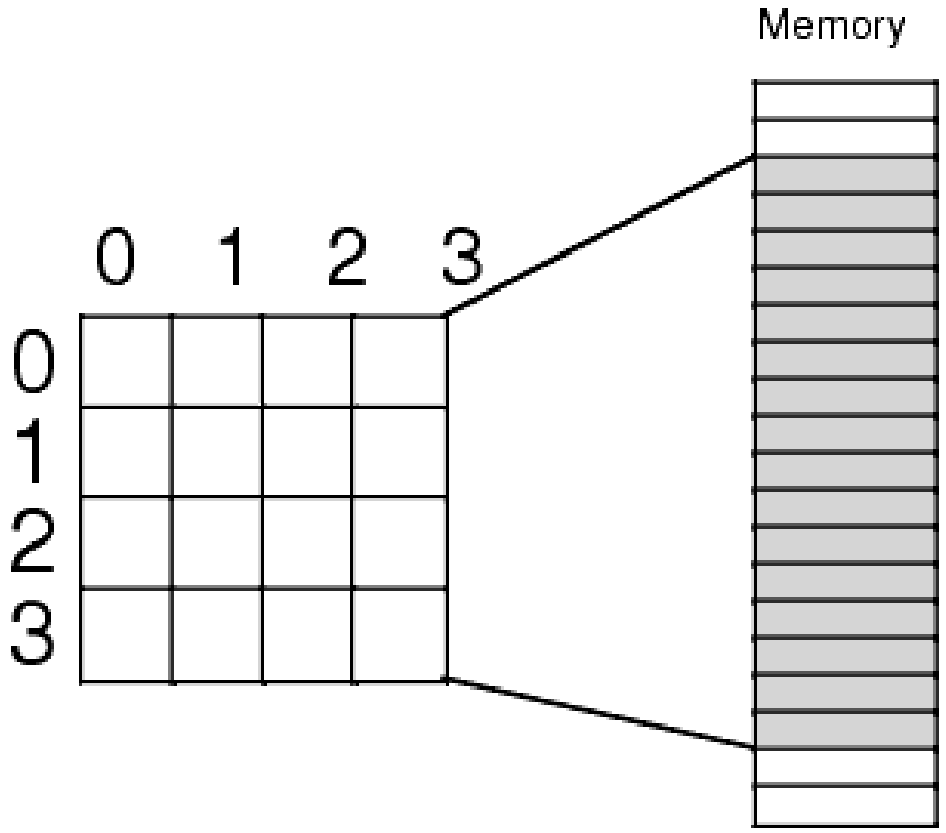
Данни на една програма

- Масив;
- Списък (свързан списък);
- Стек;
- Опашка;
- Граф;
- Дърво;

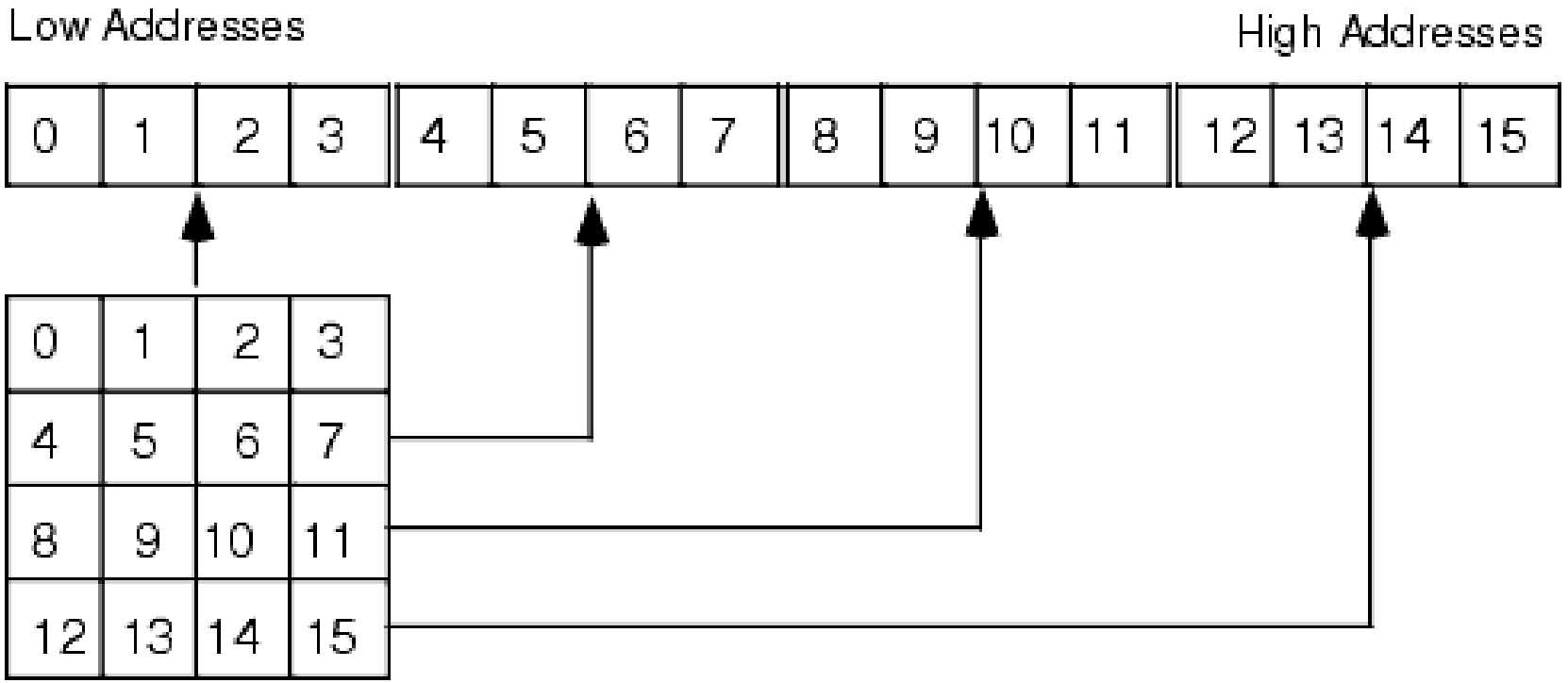
Масив

- Линейна структура;
- Последователна памет;
- Операция индексиране определя адрес на клетка (мн. бързо);

Многомерни масиви

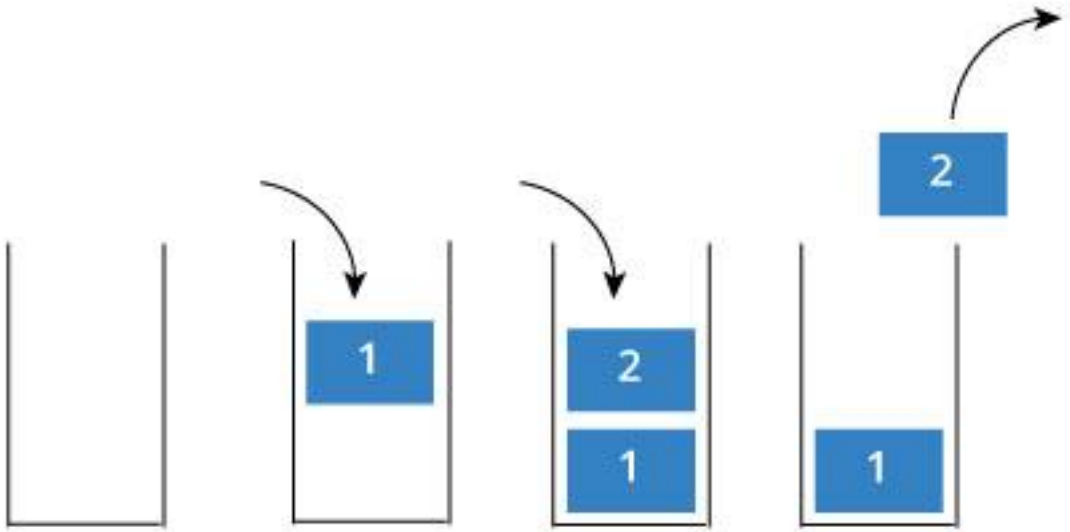


Многомерни масиви



Стек

- Линейна структура;
- **L**ast **I**n **F**irst **O**ut



Стек – статична реализация

```
class Stack
{
    const int MAX = 1000;
    int top;
    int[] stack = new int[MAX];

    public Stack()
    {
        top = -1;
    }

}
```

Стек – статична реализация – четене

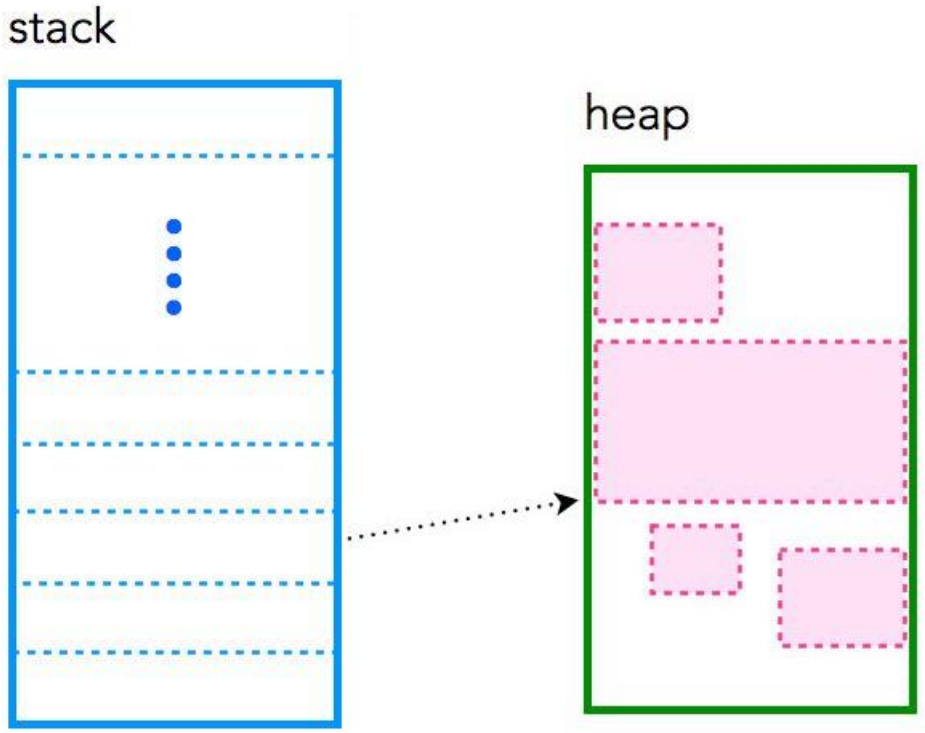
```
public int Pop()
{
    if (top < 0)
    {
        Console.WriteLine("Stack Underflow");
        return 0;
    }
    else
    {
        int value = stack[top--];
        return value;
    }
}
```

Стек – статична реализация – добавяне

```
public bool Push(int data)
{
    if (top >= MAX)
    {
        Console.WriteLine("Stack Overflow");
        return false;
    }
    else
    {
        stack[++top] = data;
        return true;
    }
}
```

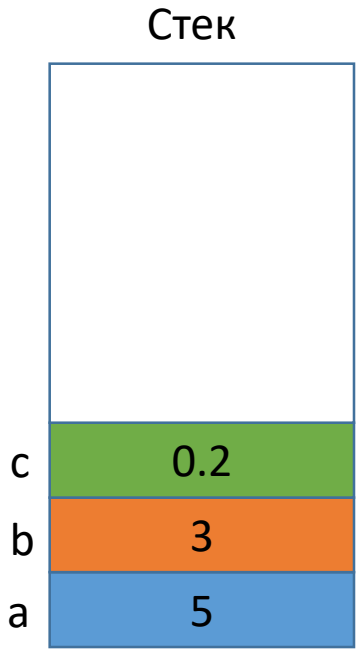
Разположение на данните

- Статично;
- Динамично.



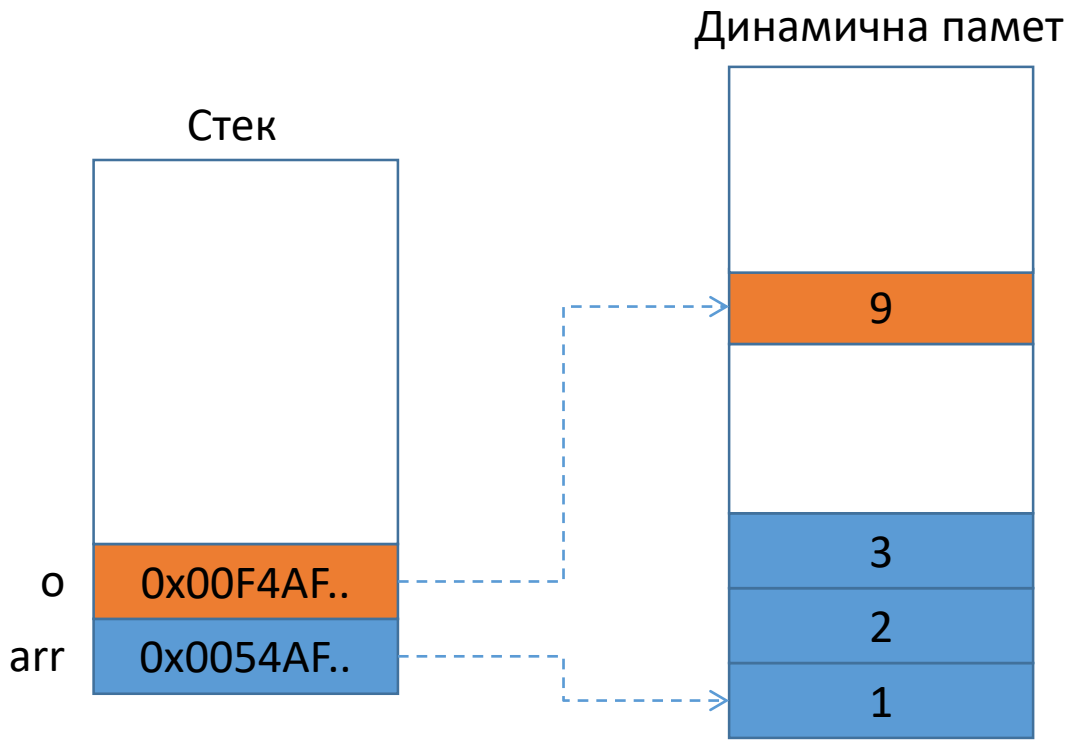
Стойностни типове (value types)

```
static void Main(string[] args)
{
    int a = 5;
    int b = 3;
    double c = 0.2;

    //...
}
```



Референтни типове (reference types)

```
static void Main(string[] args)
{
    int[] arr = new int[] {1,2,3};
    object o = 9;

    //...
}
```



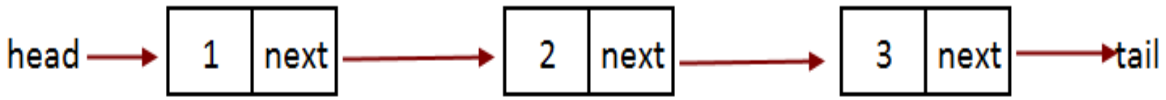
Intermediate Language

```
var x = 5;
var y = 10;
var z = 18;

var res = (x * x + 2 * y) / (z + 2);
```

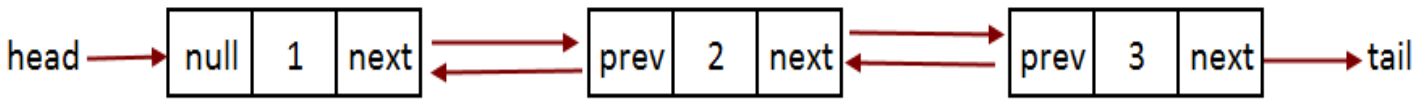
```
ldc.i4.5
stloc.0
ldc.i4.s   10
stloc.1
ldc.i4.s   18
stloc.2
ldloc.0
ldloc.0
mul
ldc.i4.2
ldloc.1
mul
add
ldloc.2
ldc.i4.2
add
div
```

Свързан списък

- Линейна структура;
- Едностранно свързан;



- Двустранно свързан.



Едностранно свързан списък

```
class LinkedList
{
    public class Node
    {
        public int data;
        public Node next;
        public Node(int d)
        {
            data = d;
            next = null;
        }
    }

    Node head;
}
```

Едностранно свързан списък – добавяне

```
public void InsertAfter(Node prev_node, int new_data)
{
    if (prev_node == null)
    {
        Console.WriteLine("The given previous node cannot be null");
        return;
    }

    Node new_node = new Node(new_data);

    new_node.next = prev_node.next;

    prev_node.next = new_node;
}
```

Едностранно свързан списък – добавяне

```
public void Append(int new_data)
{
    Node new_node = new Node(new_data);

    if (head == null)
    {
        head = new Node(new_data);
        return;
    }

    new_node.next = null;

    Node last = head;
    while (last.next != null)
        last = last.next;

    last.next = new_node;
    return;
}
```

Едностранно свързан списък – изтриване

```
void Remove(int data)
{
    Node temp = head, prev = null;

    if (temp != null && temp.data == data)
    {
        head = temp.next;
        return;
    }

    while (temp != null && temp.data != data)
    {
        prev = temp;
        temp = temp.next;
    }

    if (temp == null)
        return;

    prev.next = temp.next;
}
```

Двустранно свързан списък

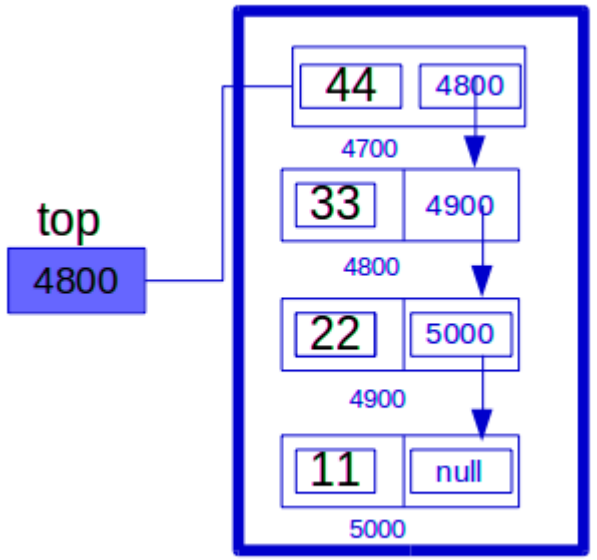
```
public class LinkedList
{
    Node head;

    class Node
    {
        int data;
        Node prev;
        Node next;

        Node(int d)
        {
            data = d;
        }
    }
}
```

Двустранно свързан списък – изтриване

```
public void Insert(Node prev_Node, int new_data)
{
    if (prev_Node == null)
    {
        Console.WriteLine("The given previous node cannot be NULL ");
        return;
    }

    Node new_node = new Node(new_data);

    new_node.next = prev_Node.next;

    prev_Node.next = new_node;

    new_node.prev = prev_Node;

    if (new_node.next != null)
        new_node.next.prev = new_node;
}
```

Стек – динамична реализация

```
public class Stack
{
    private class Node
    {
        public int data;
        public Node link;
    }

    Node top;

    public Stack()
    {
        this.top = null;
    }
}
```



Стек – динамична реализация – добавяне

```
public void Push(int x)
{
    Node temp = new Node();

    if (temp == null)
    {
        Console.Write("\nHeap Overflow");
        return;
    }

    temp.data = x;
    temp.link = top;
    top = temp;
}
```

Стек – динамична реализация – четене

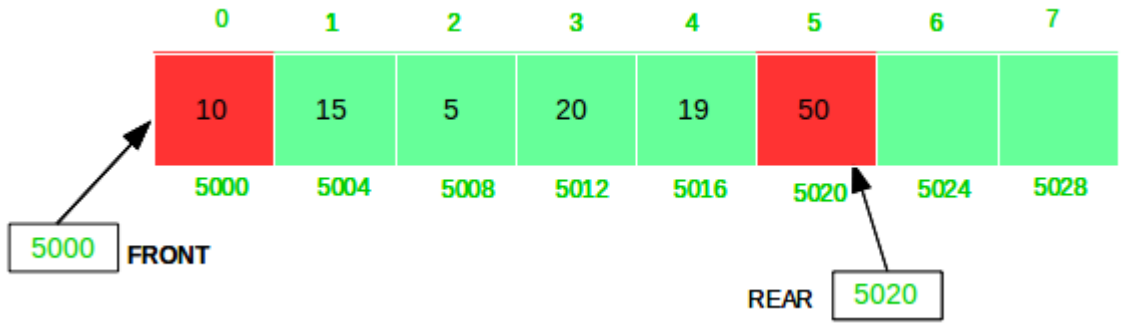
```
public void Pop()
{
    if (top == null)
    {
        Console.Write("\nStack Underflow");
        return;
    }

    top = (top).link;
}
```

Опашка

- Линейна структура;
- **F**irst **I**n **F**irst **O**ut

Опашка – статична реализация

```
class Queue
{
    private int front, rear, capacity;
    private int []queue;

    0 references
    public Queue(int c)
    {
        front = rear = 0;
        capacity = c;
        queue = new int[capacity];
    }
}
```



Опашка – статична реализация – четене

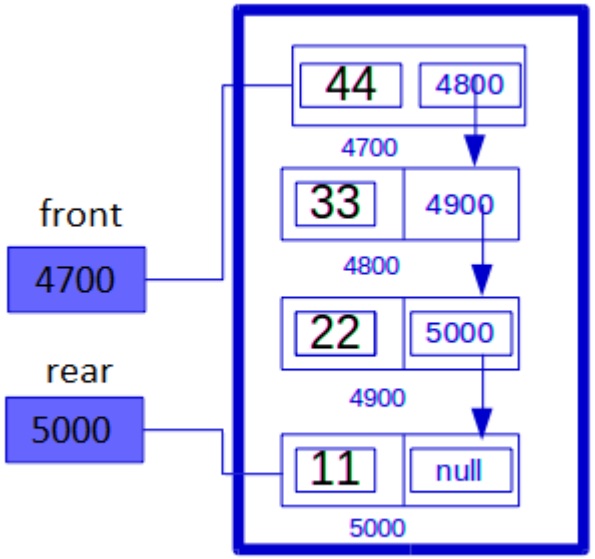
```
public void Dequeue()
{
    if (front == rear)
    {
        Console.Write("\nQueue is empty\n");
        return;
    }
    else
    {
        for (int i = 0; i < rear - 1; i++)
        {
            queue[i] = queue[i + 1];
        }

        if (rear < capacity)
            queue[rear] = 0;

        rear--;
    }
}
```

Опашка – статична реализация – писане

```
public void Enqueue(int data)
{
    if (capacity == rear)
    {
        Console.Write("\nQueue is full\n");
        return;
    }
    else
    {
        queue[rear] = data;
        rear++;
    }
}
```

Опашка – динамична реализация

```
class Queue
{
    class Node
    {
        public int key;
        public Node next;

        public Node(int key)
        {
            this.key = key;
            this.next = null;
        }
    }

    Node front, rear;

    public Queue()
    {
        this.front = this.rear = null;
    }
}
```



Опашка – динамична реализация – четене

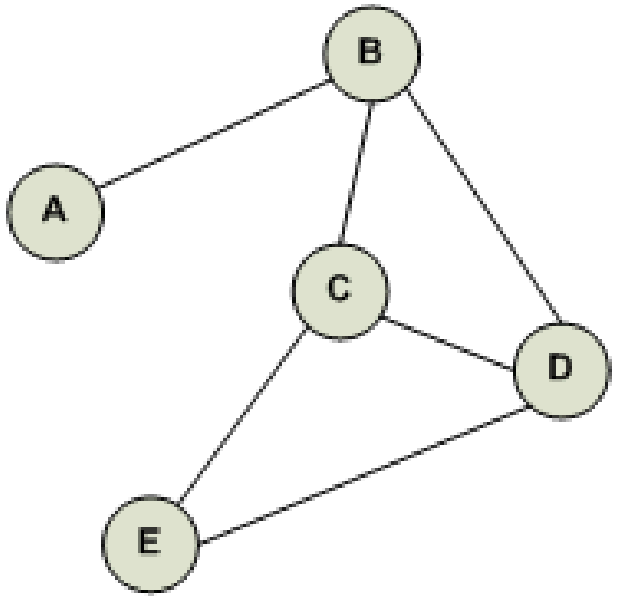
```
public Node Dequeue()
{
    if (this.front == null)
        return null;

    Node temp = this.front;
    this.front = this.front.next;

    if (this.front == null)
        this.rear = null;

    return temp;
}
```

Опашка – динамична реализация – писане

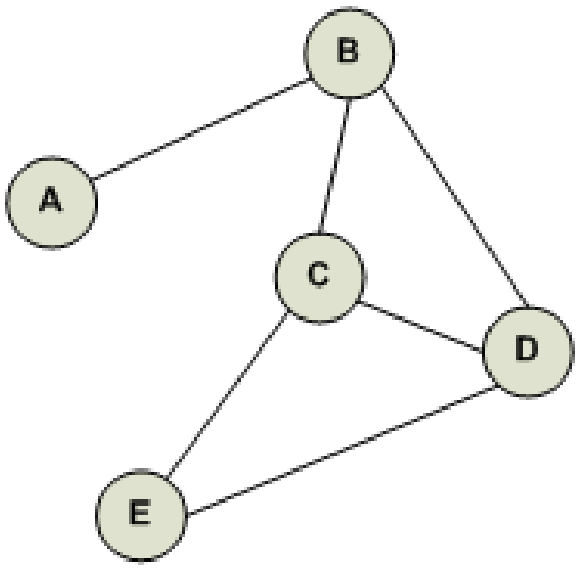
```
public void Enqueue(int key)
{
    Node temp = new Node(key);

    if (this.rear == null)
    {
        this.front = this.rear = temp;
        return;
    }

    this.rear.next = temp;
    this.rear = temp;
}
```

Граф

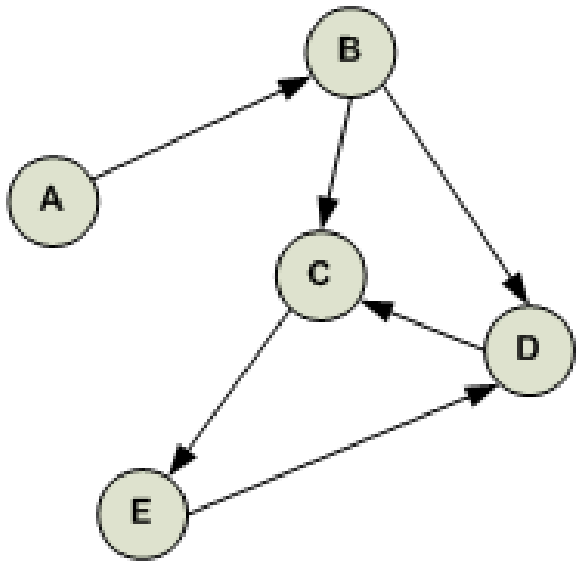
- Нелинейна структура;



Граф

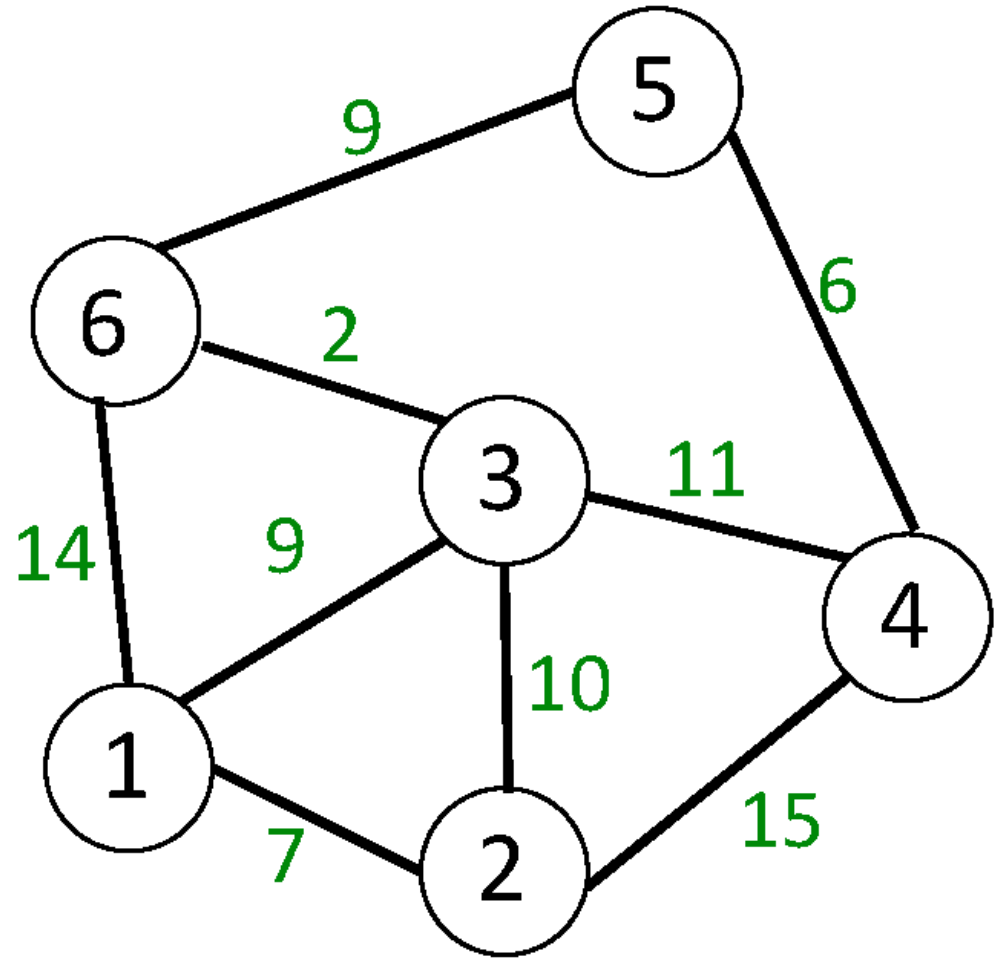


Ненасочен



Насочен

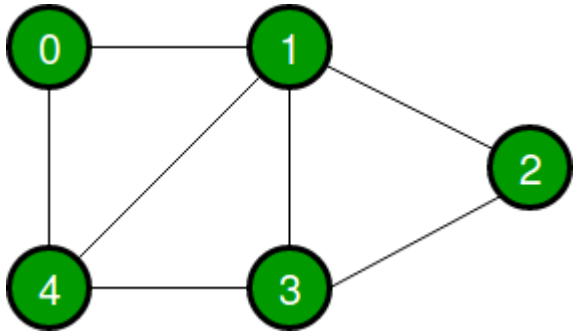
Граф



Теглови

Граф – статична реализация

```
public class Graph
{
    private int[,] nodes;

    public Graph(int c)
    {
        nodes = new int[c, c];
    }
}
```



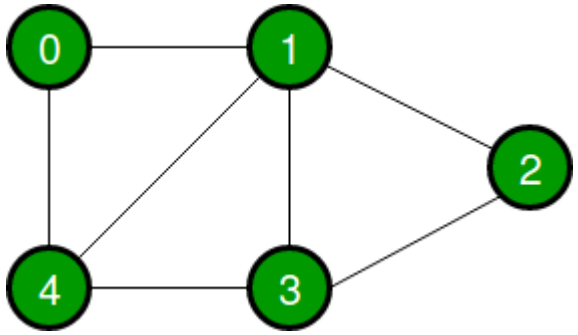
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

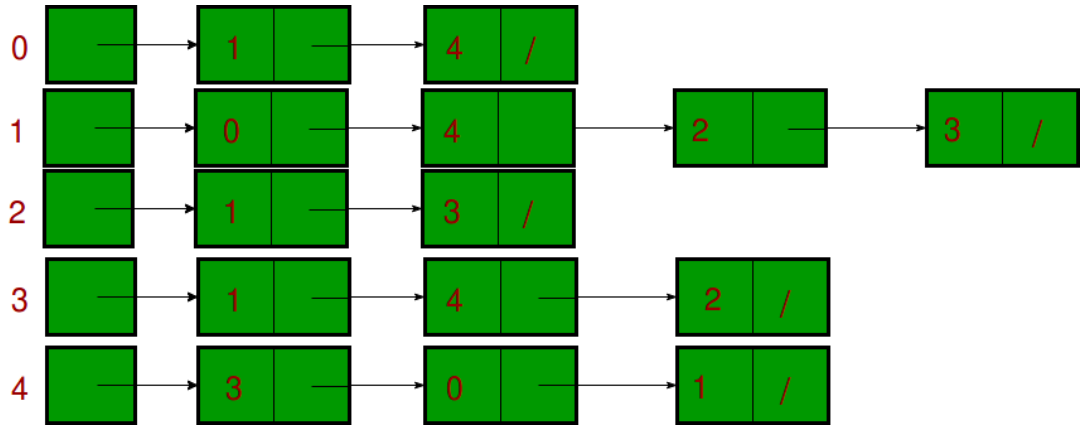
Граф – статична реализация – писане

```
public void Connect(int n1, int n2)
{
    nodes[n1, n2] = nodes[n2, n1] = 1;
}
public void Disconnect(int n1, int n2)
{
    nodes[n1, n2] = nodes[n2, n1] = 0;
}
```

Граф – динамична реализация

```
public class Graph
{
    public class Node
    {
        public int n;
        public Node next;
    }

    private Node[] nodes;
    public Graph(int c)
    {
        nodes = new Node[c];
    }
}
```





Граф – динамична реализация – писане

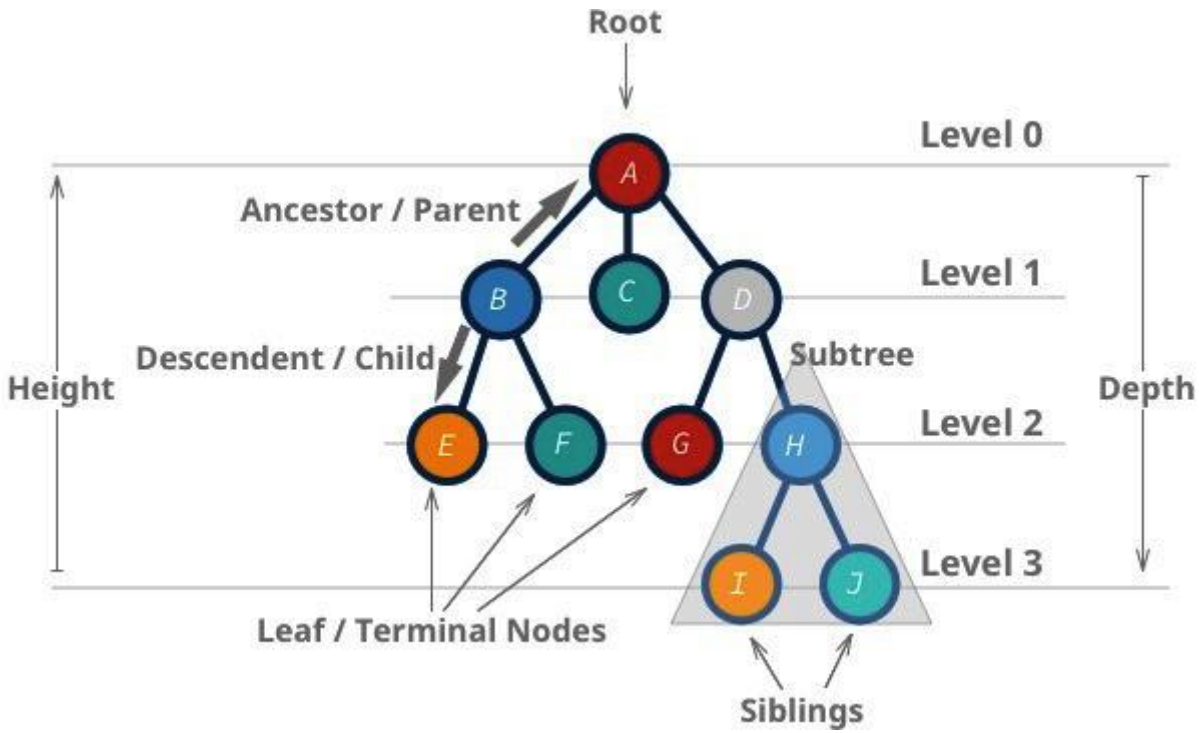
```
public void Connect(int n1, int n2)
{
    var newNode2 = new Node();
    newNode2.n = n2;
    newNode2.next = nodes[n1];
    nodes[n1] = newNode2;

    // -/- за newNode1
}
```

```
public void Disconnect(int n1, int n2)
{
    if(nodes[n1] == null)
        return;

    if(nodes[n1].n == n2)
        nodes[n1] = null;

    var current2 = nodes[n1].next;
    var current2Prev = nodes[n1];
    while (current2 != null && current2.n != n2)
    {
        current2Prev = current2;
        current2 = current2.next;
    }

    if (current2 != null)
        current2Prev.next = current2.next;

    // -/- за current1
}
```

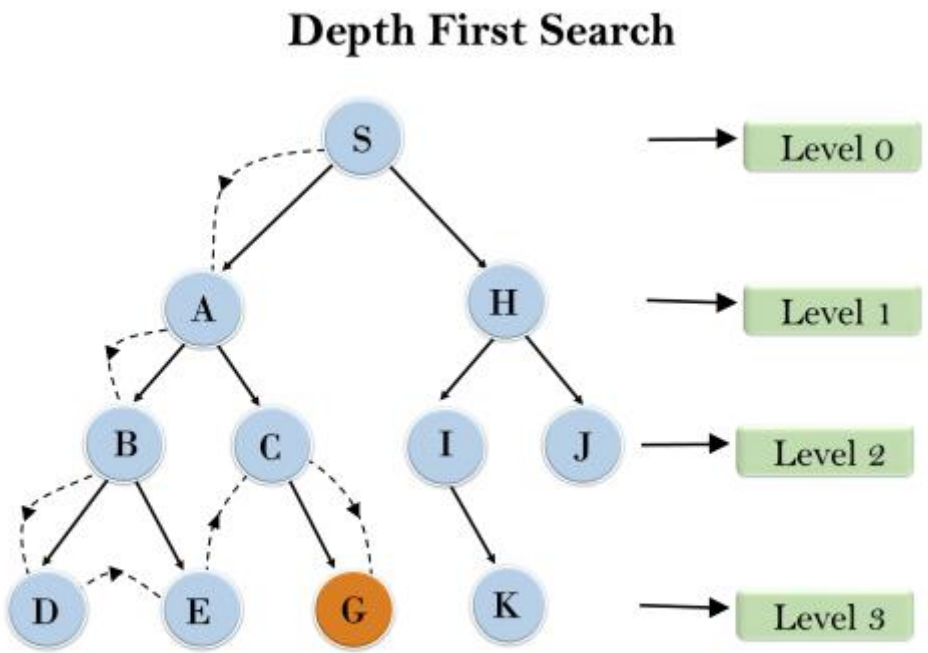
Дърво – динамична реализация (частен случай на граф)

```
public class GraphNode
{
    public List<GraphNode> connectedNodes;
}
```

Дървета – основни понятия



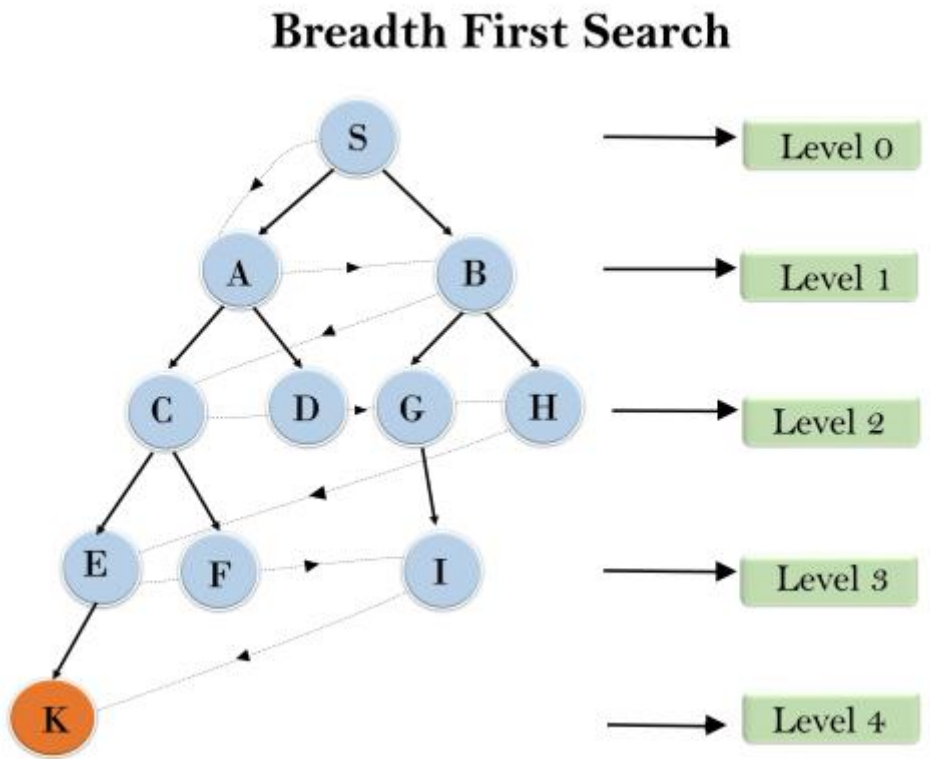
Дърво – търсене в дълбочина



Дърво – търсене в дълбочина

```
public static GraphNode DFS(GraphNode root, int n)
{
    if (root == null)
        return null;

    if (n == root.n)
        return root;

    GraphNode node = null;
    for (int i = 0; i < root.connectedNodes.Count && node == null; i++)
    {
        node = DFS(root.connectedNodes[i], n);
    }

    return node;
}
```

Дърво – търсене в широчина



Дърво – търсене в широчина

```
public static GraphNode BFS(IEnumerable<GraphNode> nodes, int n)
{
    if (nodes.Count() == 0)
        return null;

    var nextLevelNodes = new List<GraphNode>();
    foreach (var node in nodes)
        if (node.n == n)
            return node;
        else
            nextLevelNodes.AddRange(node.connectedNodes);

    return BFS(nextLevelNodes, n);
}
```

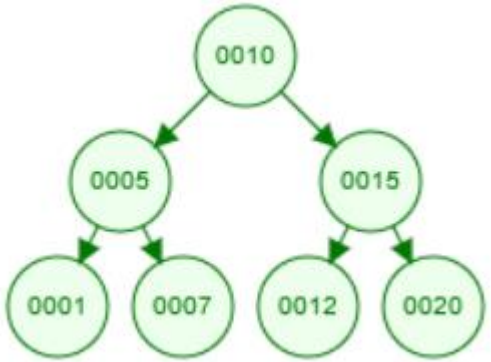
Двоично дърво

- Всеки възел има 2 подвъзела (деца): ляв и десен

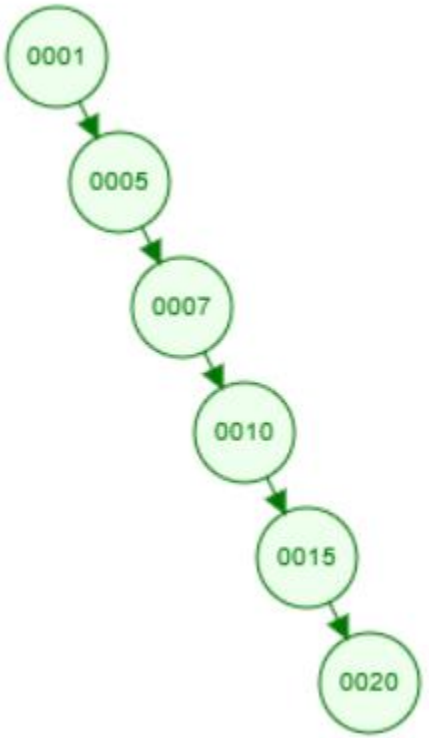
Двоично дърво за търсене

- Всеки възел има ключ и стойност;
- Ключът на всеки елемент е стойност по-голяма от ключовете на всички елементи на лявото поддърво и по-малка от ключовете на всички елементи от дясното;
- Търсенето на елемент по ключ е със сложност $O(h)$, където h е височината на дървото;
- Недостатък е, че дървото може да е небалансирано.

Двоично дърво за търсене



- Идеално балансирано (разликата между височината на лявото и дясното поддървета е не повече от 1): $h = \log_2 n$



- Небалансирано (най-лош случай): $h = n$

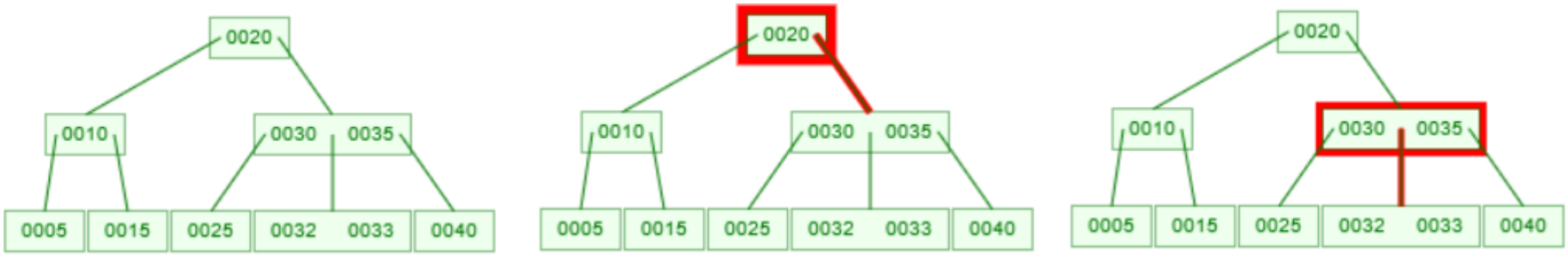
B-Tree

- Самобалансиращи се дървета – операциите добавяне и изтриване поддържат дървото в идеално балансиран вид (дължината на път от корена до всеки от възлите е една и съща);
- Използват се за търсене;
- Намират широко приложение в реализацията на бази от данни, речници и др.

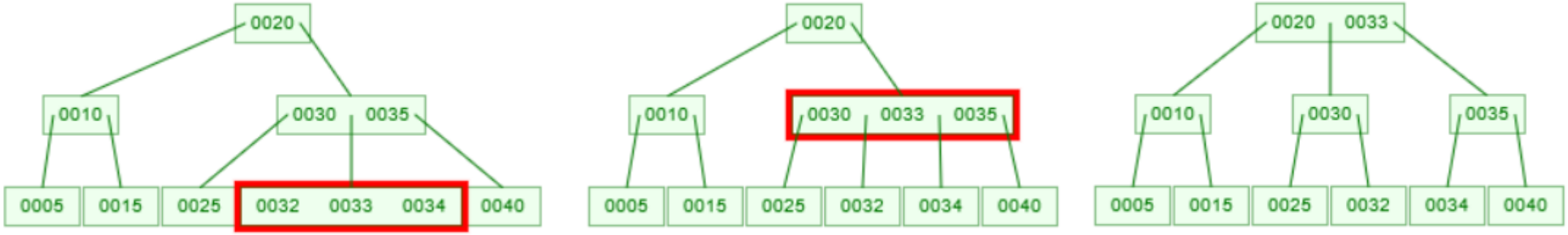
B-Tree дефиниции

- Всеки връх съдържа $k-1$ елемента (двойки ключове + данни) с ключове $t_1 < t_2 < \cdots < t_{k-1}$ и с k поддърветата $\mathrm{T}_1, \mathrm{T}_2, \ldots \mathrm{T}_k$;
- Всички ключове в поддървото T_i са по-малки от t_i и всички ключове в поддървото T_{i+1} са по-големи от t_i;
- Всеки връх освен корена и листата е връх, за който $k \in [\frac{m}{2}; m]$, където m определя от кой ред е дървото (колко поддървета може да има възел).

Добавяне на елемент с ключ „33“ в B-Tree от ред 3



Търсене на възел за новия елемент



Привеждане в балансиран вид