Бинарное дерево поиска - это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. Вывод в консоль

Нода

```
public class Node {
       private int value; // ключ узла
          private Node leftChild; // Левый узел потомок
          private Node rightChild; // Правый узел потомок

          public void printNode() { // Вывод значения узла в консоль
              System.out.println(" Выбранный узел имеет значение :" + value);
          }

          public int getValue() {
              return this.value;
          }

          public void setValue(final int value) {
              this.value = value;
          }

          public Node getLeftChild() {
              return this.leftChild;
          }

          public void setLeftChild(final Node leftChild) {
              this.leftChild = leftChild;
          }

          public Node getRightChild() {
              return this.rightChild;
          }

          public void setRightChild(final Node rightChild) {
              this.rightChild = rightChild;
          }

          @Override
          public String toString() {
              return "Node{" +
                      "value=" + value +
                      ", leftChild=" + leftChild +
                      ", rightChild=" + rightChild +
                      '}';
          }
}
```

Дерево

```java
import java.util.Stack;

public class Tree {
	private Node rootNode; // корневой узел

	    public Tree() { // Пустое дерево
	        rootNode = null;
	    }

	    public Node findNodeByValue(int value) { // поиск узла по значению
	        Node currentNode = rootNode; // начинаем поиск с корневого узла
	        while (currentNode.getValue() != value) { // поиск покуда не будет
найден элемент или не будут перебраны все
	            if (value < currentNode.getValue()) { // движение влево?
	                currentNode = currentNode.getLeftChild();
	            } else { //движение вправо
	                currentNode = currentNode.getRightChild();
	            }
	            if (currentNode == null) { // если потомка нет,
	                return null; // возвращаем null
	            }
	        }
	        return currentNode; // возвращаем найденный элемент
	    }

	    public void insertNode(int value) { // метод вставки нового элемента
	        Node newNode = new Node(); // создание нового узла
	        newNode.setValue(value); // вставка данных
	        if (rootNode == null) { // если корневой узел не существует
	            rootNode = newNode;// то новый элемент и есть корневой узел
	        }
	        else { // корневой узел занят
	            Node currentNode = rootNode; // начинаем с корневого узла
	            Node parentNode;
	            while (true) // мы имеем внутренний выход из цикла
	            {
	                parentNode = currentNode;
	                if(value == currentNode.getValue()) {   // если такой элемент в
дереве уже есть, не сохраняем его
	                    return;    // просто выходим из метода
	                 }
	                 else  if (value < currentNode.getValue()) {   // движение
влево?
	                    currentNode = currentNode.getLeftChild();
	                    if (currentNode == null){ // если был достигнут конец
цепочки,
	                        parentNode.setLeftChild(newNode); //  то вставить слева
и выйти из методы
	                        return;
	                    }
	                }
	                else { // Или направо?
	                    currentNode = currentNode.getRightChild();
	                    if (currentNode == null) { // если был достигнут конец
цепочки,
	                        parentNode.setRightChild(newNode);  //то вставить
справа
	                        return; // и выйти
	                    }
	                }
	            }
	        }
```

```
        }

        public boolean deleteNode(int value) // Удаление узла с заданным ключом
        {
            Node currentNode = rootNode;
            Node parentNode = rootNode;
            boolean isLeftChild = true;
            while (currentNode.getValue() != value) { // начинаем поиск узла
                parentNode = currentNode;
                if (value < currentNode.getValue()) { // Определяем, нужно ли
движение влево?
                    isLeftChild = true;
                    currentNode = currentNode.getLeftChild();
                }
                else { // или движение вправо?
                    isLeftChild = false;
                    currentNode = currentNode.getRightChild();
                }
                if (currentNode == null)
                    return false; // узел не найден
            }

            if (currentNode.getLeftChild() == null && currentNode.getRightChild()
== null) { // узел просто удаляется, если не имеет потомков
                if (currentNode == rootNode) // если узел - корень, то дерево
очищается
                    rootNode = null;
                else if (isLeftChild)
                    parentNode.setLeftChild(null); // если нет - узел
отсоединяется, от родителя
                else
                    parentNode.setRightChild(null);
            }
            else if (currentNode.getRightChild() == null) { // узел заменяется
левым поддеревом, если правого потомка нет
                if (currentNode == rootNode)
                    rootNode = currentNode.getLeftChild();
                else if (isLeftChild)
                    parentNode.setLeftChild(currentNode.getLeftChild());
                else
                    parentNode.setRightChild(currentNode.getLeftChild());
            }
            else if (currentNode.getLeftChild() == null) { // узел заменяется
правым поддеревом, если левого потомка нет
                if (currentNode == rootNode)
                    rootNode = currentNode.getRightChild();
                else if (isLeftChild)
                    parentNode.setLeftChild(currentNode.getRightChild());
                else
                    parentNode.setRightChild(currentNode.getRightChild());
            }
            else { // если есть два потомка, узел заменяется преемником
                Node heir = receiveHeir(currentNode);// поиск преемника для
удаляемого узла
                if (currentNode == rootNode)
                    rootNode = heir;
                else if (isLeftChild)
                    parentNode.setLeftChild(heir);
                else
                    parentNode.setRightChild(heir);
            }
            return true; // элемент успешно удалён
```

```
        }

        // метод возвращает узел со следующим значением после передаваемого
аргументом.
        // для этого он сначала переходим к правому потомку, а затем
        // отслеживаем цепочку левых потомков этого узла.
        private Node receiveHeir(Node node) {
            Node parentNode = node;
            Node heirNode = node;
            Node currentNode = node.getRightChild(); // Переход к правому потомку
            while (currentNode != null) // Пока остаются левые потомки
            {
                parentNode = heirNode;// потомка задаём как текущий узел
                heirNode = currentNode;
                currentNode = currentNode.getLeftChild(); // переход к левому
потомку
            }
            // Если преемник не является
            if (heirNode != node.getRightChild()) // правым потомком,
            { // создать связи между узлами
                parentNode.setLeftChild(heirNode.getRightChild());
                heirNode.setRightChild(node.getRightChild());
            }
            return heirNode;// возвращаем приемника
        }

        public void printTree() { // метод для вывода дерева в консоль
            Stack globalStack = new Stack(); // общий стек для значений дерева
            globalStack.push(rootNode);
            int gaps = 32; // начальное значение расстояния между элементами
            boolean isRowEmpty = false;
            String separator = "------------------------------------------------
--------------";
            System.out.println(separator);// черта для указания начала нового
дерева
            while (isRowEmpty == false) {
                Stack localStack = new Stack(); // локальный стек для задания
потомков элемента
                isRowEmpty = true;

                for (int j = 0; j < gaps; j++)
                    System.out.print(' ');
                while (globalStack.isEmpty() == false) { // покуда в общем стеке
есть элементы
                    Node temp = (Node) globalStack.pop(); // берем следующий, при
этом удаляя его из стека
                    if (temp != null) {
                        System.out.print(temp.getValue()); // выводим его значение
в консоли
                        localStack.push(temp.getLeftChild()); // сохраняем в
локальный стек, наследники текущего элемента
                        localStack.push(temp.getRightChild());
                        if (temp.getLeftChild() != null ||
                                temp.getRightChild() != null)
                            isRowEmpty = false;
                    }
                    else {
                        System.out.print("__");// - если элемент пустой
                        localStack.push(null);
                        localStack.push(null);
                    }
                    for (int j = 0; j < gaps * 2 - 2; j++)
```

```
                    System.out.print(' ');
                }
                System.out.println();
                gaps /= 2;// при переходе на следующий уровень расстояние между
элементами каждый раз уменьшается
                while (localStack.isEmpty() == false)
                    globalStack.push(localStack.pop()); // перемещаем все элементы
из локального стека в глобальный
            }
            System.out.println(separator);// подводим черту
        }
}
```

Приложение

```
public class Application {
	public static void main(String[] args) {
		Tree tree = new Tree();
		// вставляем узлы в дерево:
		tree.insertNode(6);
		tree.insertNode(8);
		tree.insertNode(5);
		tree.insertNode(8);
		tree.insertNode(2);
		tree.insertNode(9);
		tree.insertNode(7);
		tree.insertNode(4);
		tree.insertNode(10);
		tree.insertNode(3);
		tree.insertNode(1);
		// отображение дерева:
		tree.printTree();

		// удаляем один узел и выводим оставшееся дерево в консоли
		tree.deleteNode(5);
		tree.printTree();

		// находим узел по значению и выводим его в консоли
		Node foundNode = tree.findNodeByValue(7);
		foundNode.printNode();
	}
}
```

Хеш-таблица - Хеш-таблицей называется структура данных, предназначенная для реализации ассоциативного массива, такого в котором адресация реализуется посредством хеш-функции. Хеш-функция – это функция, преобразующая ключ key в некоторый индекс i равный h(key), где h(key) – хеш-код (хеш-сумма, хеш) key. Весь процесс получения индексов хеш-таблицы называется хешированием.

Графы — это абстрактный способ представления типов отношений, например, дорог, соединяющих города, и других видов сетей. Графы состоят

из рёбер и вершин. Вершина—это точка на графе, а ребро—это то, что соединяет две точки на графе.

Обход в глубину java

```
public class Vertex {
        private String label;
        public Boolean isVisited = false;

     public Vertex(String label) {
         this.label = label;
     }

     public String getLabel() {
         return label;
     }

     public void setVisited(Boolean isVisited) {
       this.isVisited = isVisited;
     }
}
```

```
import java.util.Stack;

public class Graph {
       //массив для хранения вершин
    private Vertex[] vertexArray;
    //матрица смежности
    private int[][] matrix;
    //текущее количество вершин
    private int count;
    public Stack<Integer> stack = new Stack<>();

    public Graph(int n)
    {
        vertexArray = new Vertex[n];
        matrix = new int[n][n];
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                matrix[i][j] = 0;
    }

    public void insertVertex(String key)
    {
        Vertex v = new Vertex(key);
        vertexArray[count++] = v;
    }

    public void insertEdge(int begin, int end)
    {
        matrix[begin][end] = 1;
        matrix[end][begin] = 1;
    }

  //получение смежной непосещенной вершины
    private int getUnvisitedVertex(int vertex)
```

```
    {
        for(int i = 0; i < count; i++)
            if(matrix[vertex][i] == 1 && vertexArray[i].isVisited == false)
                return i;

        return -1;
    }

  //реализация обхода в глубину
    public void dfs(int v)
    {
        System.out.print("Выполняем обход в глубину: " + vertexArray[v].getLabel() +
" -> ");
        vertexArray[v].isVisited = true;
        stack.push(v);
        while(!stack.isEmpty())
        {
            int adjVertex = getUnvisitedVertex((int) stack.peek());
            if(adjVertex == -1)
                stack.pop();
            else{
                vertexArray[adjVertex].isVisited = true;
                System.out.print(vertexArray[adjVertex].getLabel() + " -> ");
                stack.push(adjVertex);
            }
        }

        for(Vertex vertex: vertexArray)
            vertex.setVisited(false);
    }
```

```
//реализация обхода в ширину
  public void bfs(int v)
  {
    System.out.print("Выполняем обход в ширину: " + vertexArray[v].getLabel() + " -> ");
    vertexArray[v].isVisited = true;
    int vertex;
    JQueue queue = new JQueue(100);
    queue.insert(v);
    while(!queue.isEmpty())
    {
      int currentVertex = queue.pop();

      while((vertex = getUnvisitedVertex(currentVertex)) != -1)
      {
        vertexArray[vertex].setVisited(true);
        System.out.print(vertexArray[vertex].getLabel() + " -> ");
        queue.insert(vertex);
      }
    }
  }

}
```

```
public class Main {
      public static void main(String[] args) {
            Graph graph = new Graph(100);

        graph.insertVertex("A");
        graph.insertVertex("B");
```

```
            graph.insertVertex("C");
            graph.insertVertex("D");
            graph.insertVertex("E");
            graph.insertVertex("F");

            graph.insertEdge(0, 1);
            graph.insertEdge(0, 2);
            graph.insertEdge(1, 3);
            graph.insertEdge(2, 4);
            graph.insertEdge(2, 5);

            graph.dfs(0);
        }

}
```

Сортировки:

- Быстрая сортировка

Идея алгоритма следующая:

- Необходимо выбрать опорный элемент массива, им может быть любой элемент, от этого не зависит правильность работы алгоритма;
- Разделить массив на три части, в первую должны войти элементы меньше опорного, во вторую - равные опорному и в третью - все элементы больше опорного;
- Рекурсивно выполняются предыдущие шаги, для подмассивов с меньшими и большими значениями, до тех пор, пока в них содержится больше одного элемента.

```
 using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sortirovki
{
    class QuickSort
    {
        public static Random rnd = new Random();
        static void Main(string[] args)
        {
            Console.WriteLine("Ведите количество чисел для сортировки.");
            int N = Convert.ToInt32(Console.ReadLine());
            int[] mas = new int[N];
            Stopwatch SW = new Stopwatch();

            for (int i = 0; i < mas.Length; i++)
            {
                mas[i] = rnd.Next(20);
                Console.Write(mas[i] + " ");
```

```
            }
            Console.WriteLine();

            SW.Start();
            Sorting(mas, 0, mas.Length - 1);
            SW.Stop();

            Console.WriteLine("Отсортированный массив:");
            for (int i = 0; i < mas.Length; i++)
            {
                Console.Write(mas[i] + " ");
            }

            Console.WriteLine();
            Console.WriteLine("Замеренное время в миллисекундах: " +
Convert.ToString(SW.ElapsedTicks));
            Console.ReadLine();
        }

        public static void Sorting(int[] arr, long first, long last)
        {
            //выбираем опорный элемент - средний в группе
            double p = arr[(last - first) / 2 + first];
            int temp;
            long i = first, j = last;
            while (i <= j)
            {
                //двигаемся до тех пор пока не найдем элемент в левой группе больше
опорного
                while (arr[i] < p && i <= last) ++i;
                //двигаемся до тех пор пока не найдем элемент в правой группе меньше
опорного
                while (arr[j] > p && j >= first) --j;
                //меняем найденные элементы
                if (i <= j)
                {
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                    ++i; --j;
                }
            }
            //сортируем элементы в сл левой группе
            if (j > first) Sorting(arr, first, j);
            //сортируем элементы в сл правой группе
            if (i < last) Sorting(arr, i, last);
        }

    }
}
```

- Сортировка слиянием

Сортировка слиянием (marge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько

подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sortirovki
{
    class SortSliyanie
    {
        public static Random rnd = new Random();
        static void Main(string[] args)
        {
            int[] arr = new int[9] { 11, 2, 5, 0, 4, 36, 7, 1, 54 };

            //вывод на экран несортированного массива
            foreach (int item in arr)
            {
                Console.Write(item.ToString() + " ");
            }
            Console.WriteLine("\n");
            arr = Sort(arr);

            //вывод на экран отсортированного массива
            foreach (int item in arr)
            {
                Console.Write(item.ToString() + " ");
            }
            Console.Read();
        }

        static int[] Sort(int[] buff)
        {
            //проверка длинны массива
            //если длина равна 1, то возвращаем массив,
            //так как он не нуждается в сортировке
            if (buff.Length > 1)
            {
                //массивы для хранения половинок входящего буфера
                int[] left = new int[buff.Length / 2];
                //для проверки ошибки некорректного разбиения массива,
                //в случае если длина непарное число
                int[] right = new int[buff.Length - left.Length];

                //заполнение субмассивов данными из входящего массива
                for (int i = 0; i < left.Length; i++)
                {
                    left[i] = buff[i];
                }
                for (int i = 0; i < right.Length; i++)
                {
                    right[i] = buff[left.Length + i];
                }
                //если длина субмассивов больше еденици,
                //то мы повторно (рекурсивно) вызываем функцию разбиения массива
                if (left.Length > 1)
                    left = Sort(left);
```

```
                if (right.Length > 1)
                    right = Sort(right);
                //сортировка слиянием половинок
                buff = MergeSort(left, right);
            }
            //возврат отсортированного массива
            return buff;
        }

        public static int[] MergeSort(int[] left, int[] right)
        {
            //буфер для отсортированного массива
            int[] buff = new int[left.Length + right.Length];
            //счетчики длины трех массивов
            int i = 0;  //соединенный массив
            int l = 0;  //левый массив
            int r = 0;  //правый массив
                        //сортировка сравнением элементов
            for (; i < buff.Length; i++)
            {
                //если правая часть уже использована, дальнейшее движение происходит
только в левой
                //проверка на выход правого массива за пределы
                if (r >= right.Length)
                {
                    buff[i] = left[l];
                    l++;
                }
                //проверка на выход за пределы левого массива
                //и сравнение текущих значений обоих массивов
                else if (l < left.Length && left[l] < right[r])
                {
                    buff[i] = left[l];
                    l++;
                }
                //если текущее значение правой части больше
                else
                {
                    buff[i] = right[r];
                    r++;
                }
            }
            //возврат отсортированного массива
            return buff;
        }

    }
      }
```

- Поразрядная сортировка

Рассмотрим пример работы алгоритма на входном списке 0 => 8 => 12 => 56 => 7 => 26 => 44 => 97 => 2 => 37 => 4 => 3 => 3 => 45 => 10.

Максимальное число содержит две цифры, значит, разрядность данных k=2.

Первый проход, j=1.

Распределение по первой справа цифре:

L0: 0 => 10 // все числа с первой справа цифрой 0

L1: пусто

L2: 12 => 2

L3: 3 => 3

L4: 44 => 4

L5: 45

L6: 56 => 26

L7: 7 => 97 => 37

L8: 8

L9: пусто // все числа с первой справа цифрой 9

Сборка:

соединяем списки Li один за другим

L: 0 => 10 => 12 => 2 => 3 => 3 => 44 => 4 => 45 => 56 => 26 => 7 => 97 => 37 => 8

Второй проход, j=2.

Распределение по второй справа цифре:

L0: 0 => 2 => 3 => 3 => 4 => 7 => 8

L1: 10 => 12

L2: 26

L3: 37

L4: 44 => 45

L5: 56

L6: пусто

L7: пусто

L8: пусто

L9: 97

Сборка:

соединяем списки Li один за другим

L: 0 => 2 => 3 => 3 => 4 => 7 => 8 => 10 => 12 => 26 => 37 => 44 => 45 => 56 => 97

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sortirovki
{
    class RazryadSort
    {

        public static Random rnd = new Random();
        static void Main(string[] args)
        {
            Console.WriteLine("Ведите количество чисел для сортировки.");
            int N = Convert.ToInt32(Console.ReadLine());
            int[] mas = new int[N];

            for (int i = 0; i < mas.Length; i++)
            {
                mas[i] = rnd.Next(20);
                Console.Write(mas[i] + " ");
            }
            Console.WriteLine();

            int[] Newmas = SortL(mas);

            Console.WriteLine("Отсортированный массив:");
            for (int i = 0; i < mas.Length; i++)
            {
                Console.Write(mas[i] + " ");
            }

            Console.ReadLine();
        }

        public static int[] SortL(int[] arr)
        {
            if (arr == null || arr.Length == 0)
                return arr;

            int i, j;
            var tmp = new int[arr.Length];
            for (int shift = sizeof(int) * 8 - 1; shift > -1; --shift)
            {
                j = 0;
                for (i = 0; i < arr.Length; ++i)
                {
                    var move = (arr[i] << shift) >= 0;
                    if (shift == 0 ? !move : move)
                        arr[i - j] = arr[i];
                    else
                        tmp[j++] = arr[i];
                }
                Array.Copy(tmp, 0, arr, arr.Length - j, j);
            }

            return arr;
```

```
        }
    }
}
```

- Сортировка подсчетом

Сортировка подсчетом (Counting sort) – алгоритм сортировки, который применяется при небольшом количестве разных значений элементов массива данных.

Идея алгоритма заключается в следующем:

- считаем количество вхождений каждого элемента массива;
- исходя из данных полученных на первом шаге, формируем отсортированный массив.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sortirovki
{
    class SortPodcshet
    {
        public static Random rnd = new Random();
        static void Main(string[] args)
        {
            Console.WriteLine("Ведите количество чисел для сортировки подсчетом.");
            int N = Convert.ToInt32(Console.ReadLine());
            int[] mas = new int[N];

            for (int i = 0; i < mas.Length; i++)
            {
                mas[i] = rnd.Next(20);
                Console.Write(mas[i] + " ");
            }
            Console.WriteLine();

            int[] newMas = BasicCountingSort(mas, 20);

            Console.WriteLine("Отсортированный массив:");
            for (int i = 0; i < newMas.Length; i++)
            {
                Console.Write(newMas[i] + " ");
            }

            Console.ReadLine();
        }

        static int[] BasicCountingSort(int[] array, int k)
        {
            //заполняем массив count равный k = макс число в ориг массиве
            var count = new int[k + 1];
            for (var i = 0; i < array.Length; i++)
            {
                count[array[i]]++;
            }
```

```
            var index = 0;
            //проходим по всему массиву count
            for (var i = 0; i < count.Length; i++)
            {
                //находим ненулевой элемент и переносим в ориг массив столько раз, какое
число стоит в подсчете (count)
                for (var j = 0; j < count[i]; j++)
                {
                    array[index] = i;
                    index++;
                }
            }

            return array;
        }
    }
}
```

- Гномья сортировка

Цикл стартует не с нулевого, а первого элемента массива (см. граничные условия). В зависимости от того, как проходит сравнение двух элементов мы либо меняем их местами и отходим на шаг назад, либо не меняем и шагаем на шаг вперед. Особенность этого алгоритма заключается в том, что нам здесь не потребуются вложенные циклы, как, например, в пузырьковой сортировке.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sortirovki
{
    class GnomSort
    {
        public static Random rnd = new Random();
        static void Main(string[] args)
        {
            Console.WriteLine("Ведите количество чисел для сортировки подсчетом.");
            int N = Convert.ToInt32(Console.ReadLine());
            int[] mas = new int[N];

            for (int i = 0; i < mas.Length; i++)
            {
                mas[i] = rnd.Next(20);
                Console.Write(mas[i] + " ");
            }
            Console.WriteLine();

            GnomeSort(mas);

            Console.WriteLine("Отсортированный массив:");
            for (int i = 0; i < mas.Length; i++)
            {
                Console.Write(mas[i] + " ");
            }
```

```
            Console.ReadLine();
        }


        static void GnomeSort(int[] inArray)
        {
            int i = 1;
            int j = 2;
            while (i < inArray.Length)
            {
                if (inArray[i - 1] < inArray[i])
                {
                    i = j;
                    j += 1;
                }
                else
                {
                    //если тек элемент меньше пред, то меняем местами, шагаем назад
                    Swap(inArray, i - 1, i);
                    i -= 1;
                    //если шаг на начале массива, возвращаемся с указателя j
                    if (i == 0)
                    {
                        i = j;
                        j += 1;
                    }
                }
            }
        }

        static void Swap(int[] array, int i, int j)
        {
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }

    }
}
```

На C: двусвязный список: консольное приложение c++

```
#include <stdio.h>
#include <stdlib.h>

typedef struct _Node {
	void* value;
	struct _Node* next;
	struct _Node* prev;
} Node;

typedef struct _DblLinkedList {
	size_t size;
	Node* head;
	Node* tail;
} DblLinkedList;

DblLinkedList* createDblLinkedList() {
	DblLinkedList* tmp = (DblLinkedList*)malloc(sizeof(DblLinkedList));
	tmp->size = 0;
	tmp->head = tmp->tail = NULL;
```

```
	return tmp;
}

void deleteDblLinkedList(DblLinkedList** list) {
	Node* tmp = (*list)->head;
	Node* next = NULL;
	while (tmp) {
		next = tmp->next;
		free(tmp);
		tmp = next;
	}
	free(*list);
	(*list) = NULL;
}

void pushFront(DblLinkedList* list, void* data) {
	Node* tmp = (Node*)malloc(sizeof(Node));
	if (tmp == NULL) {
		exit(1);
	}
	tmp->value = data;
	tmp->next = list->head;
	tmp->prev = NULL;
	if (list->head) {
		list->head->prev = tmp;
	}
	list->head = tmp;

	if (list->tail == NULL) {
		list->tail = tmp;
	}
	list->size++;
}

void* popFront(DblLinkedList* list) {
	Node* prev;
	void* tmp;
	if (list->head == NULL) {
		exit(2);
	}

	prev = list->head;
	list->head = list->head->next;
	if (list->head) {
		list->head->prev = NULL;
	}
	if (prev == list->tail) {
		list->tail = NULL;
	}
	tmp = prev->value;
	free(prev);

	list->size--;
	return tmp;
}

void pushBack(DblLinkedList* list, void* value) {
	Node* tmp = (Node*)malloc(sizeof(Node));
	if (tmp == NULL) {
		exit(3);
	}
	tmp->value = value;
	tmp->next = NULL;
	tmp->prev = list->tail;
```

```
	if (list->tail) {
		list->tail->next = tmp;
	}
	list->tail = tmp;

	if (list->head == NULL) {
		list->head = tmp;
	}
	list->size++;
}

void* popBack(DblLinkedList* list) {
	Node* next;
	void* tmp;
	if (list->tail == NULL) {
		exit(4);
	}

	next = list->tail;
	list->tail = list->tail->prev;
	if (list->tail) {
		list->tail->next = NULL;
	}
	if (next == list->head) {
		list->head = NULL;
	}
	tmp = next->value;
	free(next);

	list->size--;
	return tmp;
}

Node* getNth(DblLinkedList* list, size_t index) {
	Node* tmp = list->head;
	size_t i = 0;

	while (tmp && i < index) {
		tmp = tmp->next;
		i++;
	}

	return tmp;
}



void insert(DblLinkedList* list, size_t index, void* value) {
	Node* elm = NULL;
	Node* ins = NULL;
	elm = getNth(list, index);
	if (elm == NULL) {
		exit(5);
	}
	ins = (Node*)malloc(sizeof(Node));
	ins->value = value;
	ins->prev = elm;
	ins->next = elm->next;
	if (elm->next) {
		elm->next->prev = ins;
	}
	elm->next = ins;

	if (!elm->prev) {
		list->head = elm;
```

```
	}
	if (!elm->next) {
		list->tail = elm;
	}

	list->size++;
}


void* deleteNth(DblLinkedList* list, size_t index) {
	Node* elm = NULL;
	void* tmp = NULL;
	elm = getNth(list, index);
	if (elm == NULL) {
		exit(5);
	}
	if (elm->prev) {
		elm->prev->next = elm->next;
	}
	if (elm->next) {
		elm->next->prev = elm->prev;
	}
	tmp = elm->value;

	if (!elm->prev) {
		list->head = elm->next;
	}
	if (!elm->next) {
		list->tail = elm->prev;
	}

	free(elm);

	list->size--;

	return tmp;
}



void printDblLinkedList(DblLinkedList* list, void(*fun)(void*)) {
	Node* tmp = list->head;
	while (tmp) {
		fun(tmp->value);
		tmp = tmp->next;
	}
	printf("\n");
}


void printInt(void* value) {
	printf("%d ", *((int*)value));
}

DblLinkedList* fromArray(void* arr, size_t n, size_t size) {
	DblLinkedList* tmp = NULL;
	size_t i = 0;
	if (arr == NULL) {
		exit(7);
	}

	tmp = createDblLinkedList();
	while (i < n) {
		pushBack(tmp, ((char*)arr + i * size));
		i++;
```

```
	}

	return tmp;
}

void main()
{
	DblLinkedList* list = createDblLinkedList();
	int a, b, c, d, e, f, g, h;

	a = 10;
	b = 20;
	c = 30;
	d = 40;
	e = 50;
	f = 60;
	g = 70;
	h = 80;
	pushFront(list, &a);
	pushFront(list, &b);
	pushFront(list, &c);
	pushBack(list, &d);
	pushBack(list, &e);
	pushBack(list, &f);
	printDblLinkedList(list, printInt);
	printf("length %d\n", list->size);

	printf("popFront %d\n", *((int*)popFront(list)));
	printDblLinkedList(list, printInt);
	printf("popBack %d\n", *((int*)popBack(list)));
	printDblLinkedList(list, printInt);

	printf("head %d\n", *((int*)(list->head->value)));
	printf("tail %d\n", *((int*)(list->tail->value)));
	printf("length %d\n", list->size);
	printDblLinkedList(list, printInt);
	insert(list, 2, &g);
	printDblLinkedList(list, printInt);


	//getch();
}
```