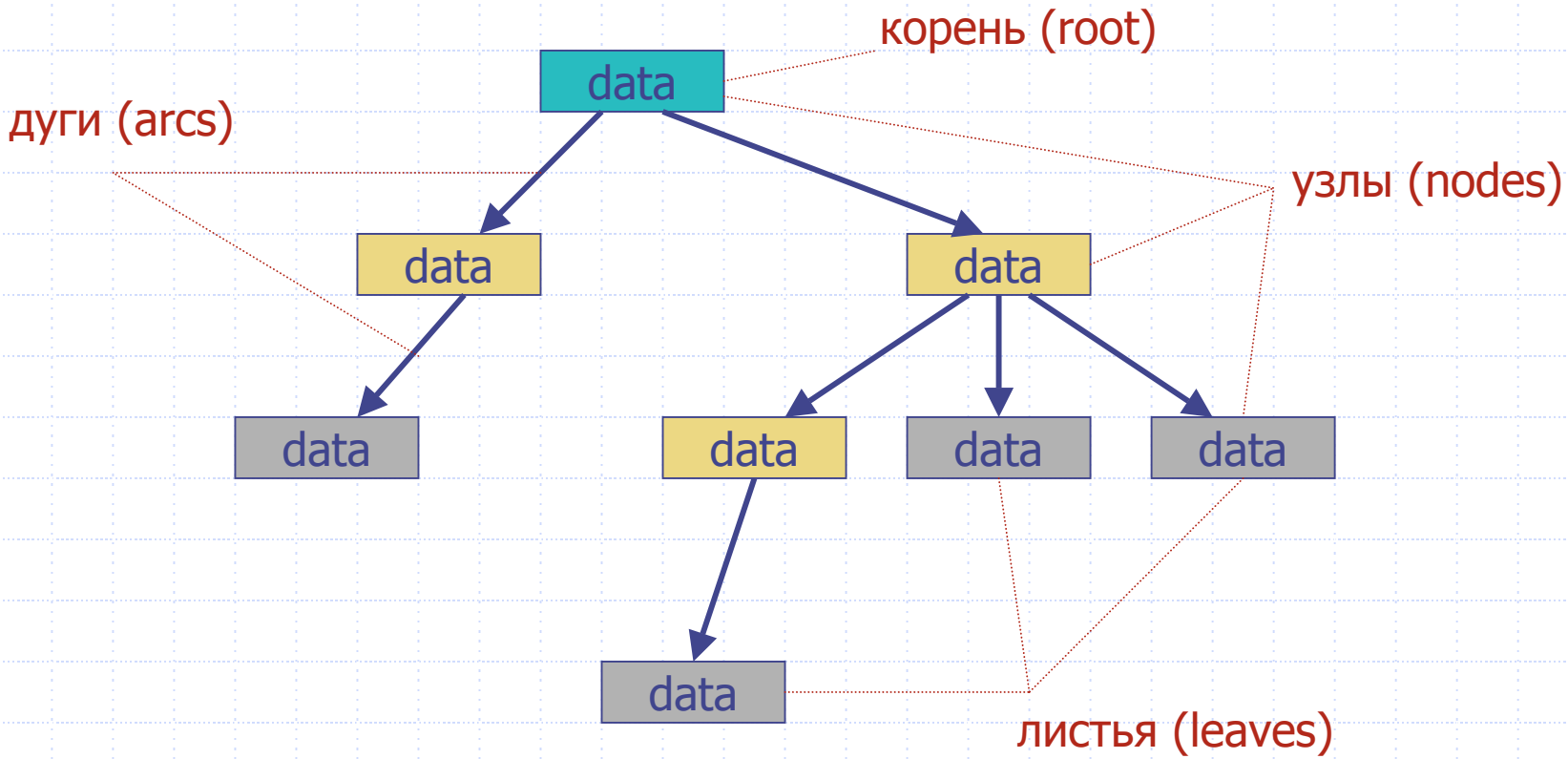
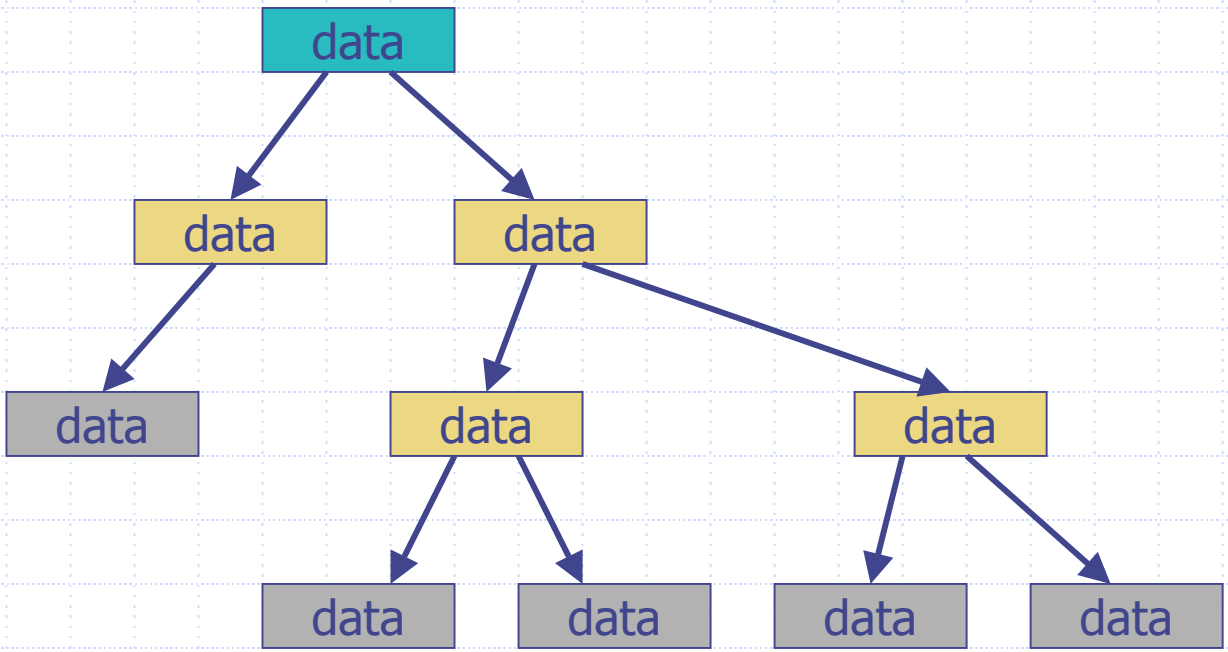
Деревья


корень (root)
дуги (arcs)
узлы (nodes)
data
data
data
data
data
data
data
data
листья (leaves)

Бинарные деревья

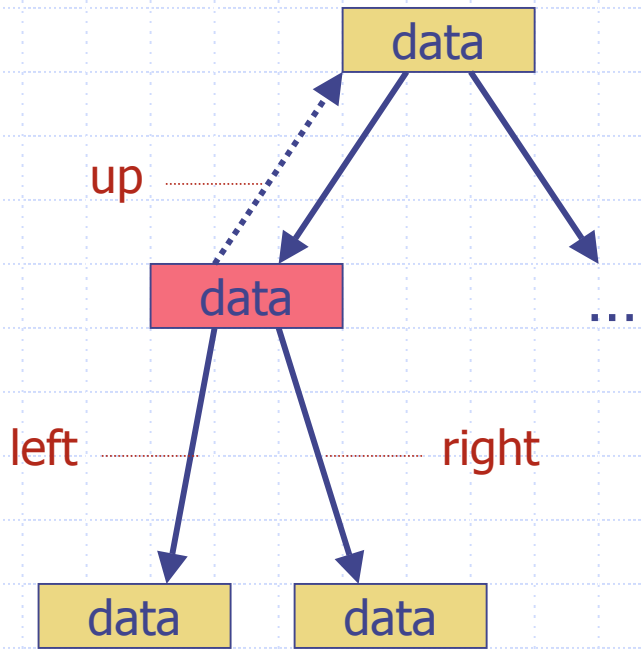
Бинарное дерево – дерево, у каждого узла которого существует не более чем два поддерева



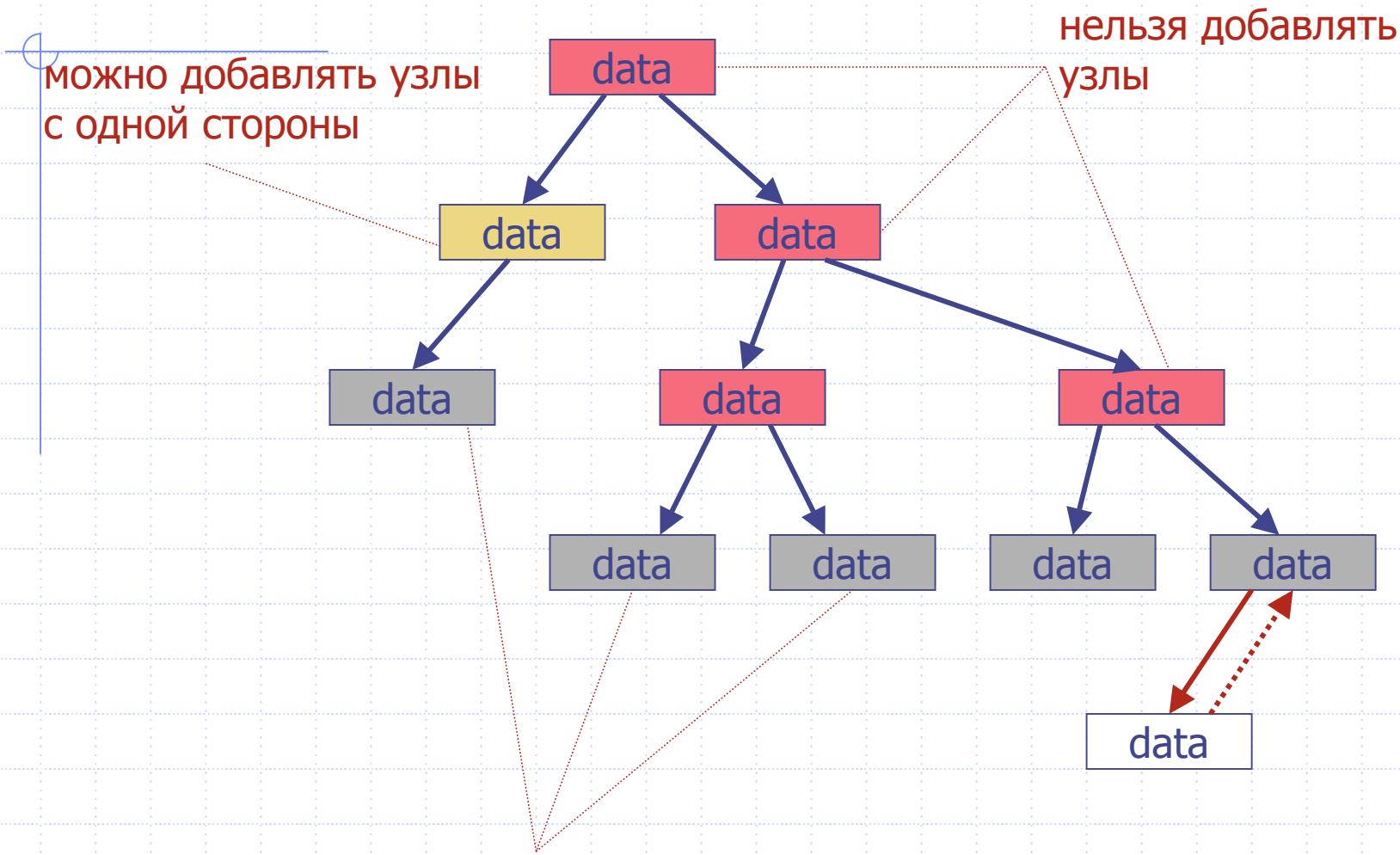
Структура “бинарное дерево”

```
typedef double TreeData;

struct Tree
{
        TreeData data;
        Tree * left, * right;
        Tree * up;
};
```



Создание узла в дереве



Создание узла в дереве

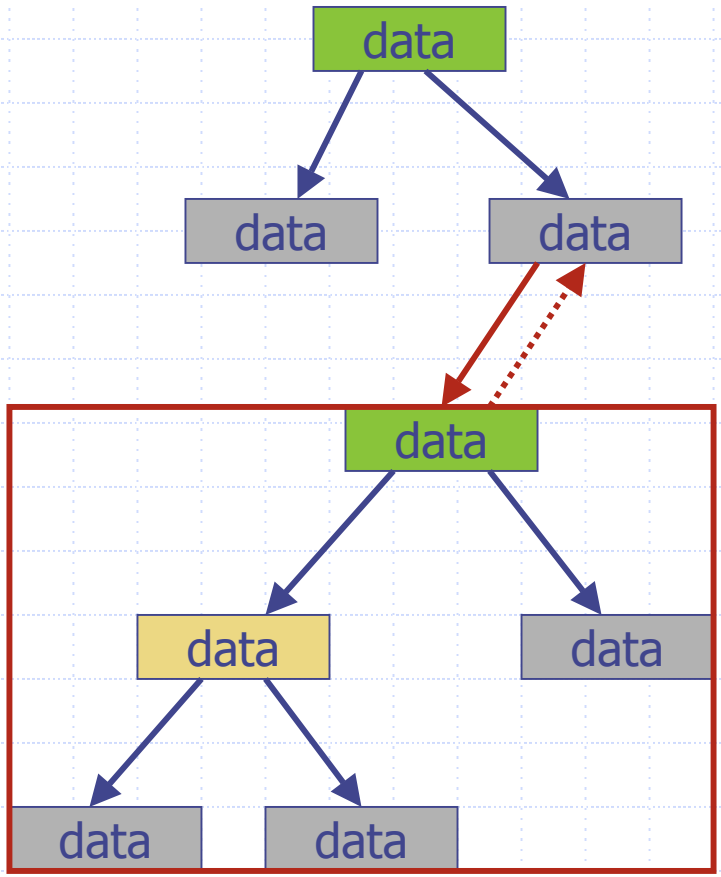
```
enum TreeAlign {Left, Right};

Tree * createNode (TreeData data, Tree * superNode, TreeAlign align)
{
        Tree * result = 0;
        if (!superNode)
        {
                //новое дерево
                //1
        }
        else if (align == Left && superNode->left == 0 ||
                align == Right && superNode->right == 0)
        {
                //добавление в существующее
                //дерево
                //2
        }
        return result;
}
```

```
//1
result = new Tree;
result->data = data;
result->up = 0;
result->left = 0;
result->right = 0;
```

```
//2
result = new Tree;
result->data = data;
result->up = superNode;
if (align == Left)
        superNode->left = result;
else superNode->right = result;
result->left = 0;
result->right = 0;
```

Добавление поддерева



Добавление поддерева

```
Tree * linkSubTree (Tree * superNode, Tree * subTree, TreeAlign align)
{
        if (!superNode || !subTree || subTree->up) return 0;
        if (align == Left && !superNode->left)
        {
                superNode->left = subTree;
                subTree->up = superNode;
        }
        else if (align == Right && !superNode->right)
        {
                superNode->right = subTree;
                subTree->up = superNode;
        }
        else return 0;
        return subTree;
}
```

Навигация по дереву

```
Tree * left (const Tree * node)
{
        return node ? node->left : 0;
}

Tree * right (const Tree * node)
{
        return node ? node->right : 0;
}

Tree * up (const Tree * node)
{
        return node ? node->up : 0;
}
```

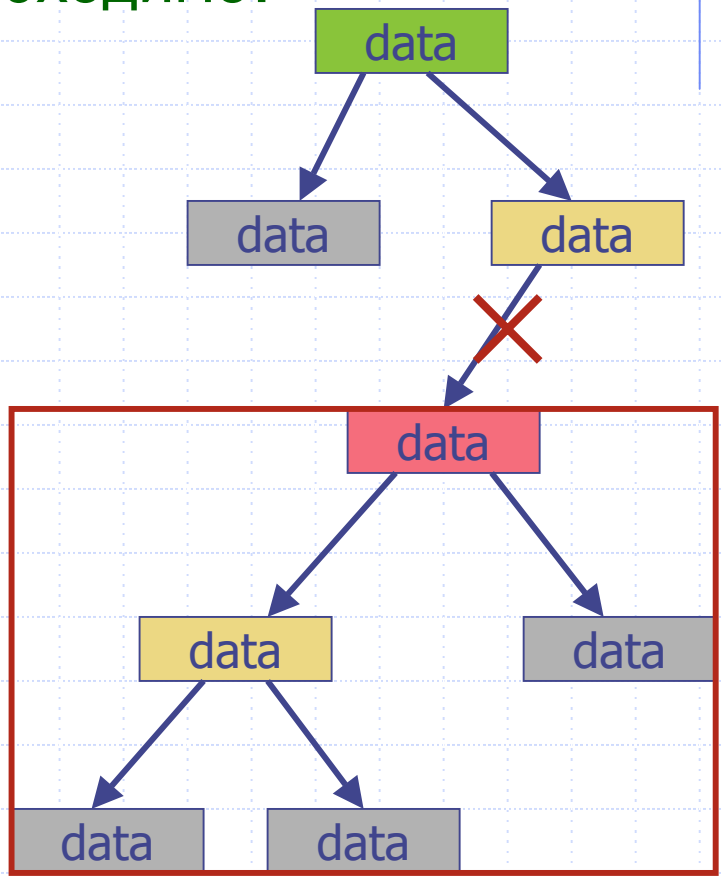
Удаление поддерева

```
Tree * cutSubTree (Tree * superNode, TreeAlign align)
{
        if (!superNode) return 0;
        Tree * subTree = 0;
        if (align == Left && superNode->left)
        {
                subTree = superNode->left;
                subTree->up = 0;
                superNode->left = 0;
        }
        else if (align == Right && superNode->right)
        {
                subTree = superNode->right;
                subTree->up = 0;
                superNode->right = 0;
        }
        return subTree;
}
```

Уничтожение поддерева

Чтобы уничтожить поддерево необходимо:

1. Уничтожить ссылку на него
2. Уничтожить левое поддерево
3. Уничтожить правое поддерево
4. Уничтожить корневой элемент



Рекурсия

Рекурсивная последовательность – такая последовательность, где каждый последующий член определяется через несколько предыдущих

Числа Фибоначчи: $a_0 = 0$, $a_1 = 1$, $a_i = a_{i-1} + a_{i-2}$

```
int fibo (int index)
{
        if (index > 1) return fibo (index – 1) + fibo (index - 2);
        else if (index == 1) return 1;
        else return 0;
}
```

Уничтожение поддерева

```
void deleteTreeRecursive (Tree * root)
{
        if (root->right) deleteTreeRecursive (root->right);
        if (root->left) deleteTreeRecursive (root->left);
        delete root;
}

void deleteSubTree (Tree * root)
{
        if (root->up)
        {
                if (root->up->left == root) root->up->left = 0;
                if (root->up->right == root) root->up->right = 0;
        }
        deleteTreeRecursive (root);
}
```

Обход дерева

Задача: обойти все узлы дерева и совершить над ними какие-либо действия (например распечатать содержимое)

Алгоритм:

1. Совершить действие над корневым узлом
2. Обойти левое поддерево
3. Обойти правое поддерево

Обход дерева

```
void traverseTree (const Tree * root, void (*action) (const Tree *))
{
    if (root == 0) return;
    action (root);
    if (root->left) traverseTree (root->left, action);
    if (root->right) traverseTree (root->right, action);
}

void printNode (const Tree * node)
{
    if (node->up == 0) printf ("(root)");
    if (!node->left && !node->right) printf ("(leaf)");
    printf ("data = %lg\n", node->data);
}

int main (){
            /*...*/ traverseTree (root, printNode); /*...*/
}
```

Указатель на функцию

Описание функции:
возвращаемый_тип имя (список_типов_параметров);

Определение указателя на функцию:
возвращаемый_тип (*имя) (список_типов_параметров);

```
...
typedef size_t (*functionType) (char const *);
functionType stringLength = strlen;
//size_t (*) (stringLength char const *) = strlen;
stringLength (str);
...
```

Обход дерева

```
//эта функция работает только для немодифицирующих действий
void traverseTree (const Tree * root, void (*action) (const Tree *));

//немодифицирующее действие
void printNode (const Tree * node);


//модифицирующее действие – нужна другая функция traverseTree
void initNode (Tree * node);

//эта функция работает только для немодифицирующих действий
void traverseTree (Tree * root, void (*action) (Tree *));
```

N-дерево: постановка задачи

Необходимо реализовать стандартный набор операций над деревом, которое обладает следующими свойствами:

1. У каждого узла может быть не более чем N поддеревьев
2. Доступ к поддеревьям осуществляется через целочисленные индексы
3. При добавлении/удалении поддеревьев в узел индексы остальных поддеревьев данного узла не меняются

Проектирование N-дерева

Цели проектирования:

1. Реализовать функциональность, заложенную в постановке задачи в виде набора функций
2. Набор функции должен быть по возможности полным – пользователю должно быть достаточно знания функции и имени структуры, но не ее полей
3. Дерево всегда должно находиться в работоспособном состоянии при условии соблюдения ограничений на использование дерева. Одним из ограничений является возможность прямого доступа только к ограниченному множеству полей

Структура NTree

```
typedef double NTreeData;

int const MaxSubNodes = 10;

struct  NTree
{
        NTree * subNodes [MaxSubNodes];
        NTree * up;
};
```

Навигация по дереву

```
//В случае ошибки возвращает 0
NTree * goUp (const NTree * node);


//В случае ошибки возвращает 0
NTree * goDown (const NTree * node, int index);
```

Добавление/удаление

```
//Возвращает указатель на новый узел
//в случае ошибки – 0
NTree * createNode (NTreeData data,
                    NTree * superNode, int index);

//Возвращает указатель на поддерево
//в случае ошибки – 0
NTree * linkSubNTree (NTree * superNode,
                    NTree * subNTree, int index);

//Возвращает указатель на поддерево
//в случае ошибки – 0
Tree * cutSubNTree (NTree * superNode,
                    int index);
```

Обход N-дерева

```
void deleteSubNTree (NTree * root);


void traverseTree (const Tree * root,
                   void (*action) (const Tree *));


void traverseTree (Tree * root,
                   void (*action) (Tree *));
```

Другие функции для N-дерева

```
NTreeData getData (const NTree * node);

void setData (NTree * node, NTreeData value);

//В случае ошибки возвращает -1
int findFreeIndex (const NTree * node);
```