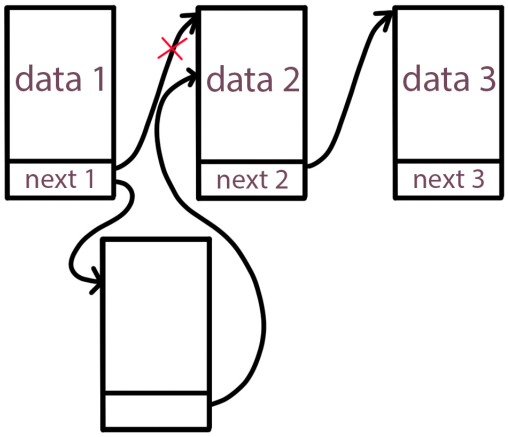
**ЦЕПНЫЕ СТРУКТУРЫ ДАННЫХ**
**- это такие СД, в которых чередующиеся элементы связаны указателями.**

**Фрагментация памяти**:
Выделение памяти происходит блоками — непрерывными фрагментами оперативной памяти (таким образом, каждый блок — это несколько идущих подряд байт). В какой-то момент в куче попросту может не оказаться блока подходящего размера и, даже если свободная память достаточна для размещения объекта, операция выделения памяти окончится неудачей.

Основной минус цепных СД: не знаем, где лежит i-й элемент → цепные СД не обладают свойством быстрого обращения.




- расширили СД за константное время.

# 1. STACK

**Стек** - СД, работающая только с последними элементами, записанными в нее по принципу **LIFO** (Last In First Out).
Пример стека: локальная память.
**Реализация:**

```
struct Node {
    int data;
    Node* next = Null;
}
struct Stack {
    Node* head=Null;
    stack(int a);
    ~stack();
    void push (int a);
    int pop ();
int multipop(int k);//deleting last k elements
}
...
```

# 2. QUEUE

В **очереди** можно добавлять элементы в конец (*tail*) и в начало (*head*).

```
// define default capacity of the queue
#define SIZE 10

// Class for queue
class queue
{
        int *arr;// array to store queue elements
        int capacity;// maximum capacity of the queue
        int front;// front points to front element in the queue
        int rear;// rear points to last element in the queue
        int count;// current size of the queue

public:
        queue(int size = SIZE);// constructor
        ~queue();// destructor

        void dequeue();
        void enqueue(int x);
        int peek();
        int size();
        bool isEmpty();
        bool isFull();
};

// Constructor to initialize queue
queue::queue(int size)
{
        arr = new int[size];
        capacity = size;
        front = 0;
        rear = -1;
        count = 0;
}

// Destructor to free memory allocated to the queue
queue::~queue()
{
        delete arr;
}

// Utility function to remove front element from the queue
void queue::dequeue()
{
        // check for queue underflow
        if (isEmpty())
        {
```

```
                cout << "UnderFlow\nProgram Terminated\n";
                exit(EXIT_FAILURE);
        }

        cout << "Removing " << arr[front] << '\n';

        front = (front + 1) % capacity;
        count--;
}

// Utility function to add an item to the queue
void queue::enqueue(int item)
{
        // check for queue overflow
        if (isFull())
        {
                cout << "OverFlow\nProgram Terminated\n";
                exit(EXIT_FAILURE);
        }

        cout << "Inserting " << item << '\n';

        rear = (rear + 1) % capacity;
        arr[rear] = item;
        count++;
}

// Utility function to return front element in the queue
int queue::peek()
{
        if (isEmpty())
        {
                cout << "UnderFlow\nProgram Terminated\n";
                exit(EXIT_FAILURE);
        }
        return arr[front];
}

// Utility function to return the size of the queue
int queue::size()
{
        return count;
}

// Utility function to check if the queue is empty or not
bool queue::isEmpty()
{
        return (size() == 0);
}
```

```
// Utility function to check if the queue is full or not
bool queue::isFull()
{
        return (size() == capacity);
}
...
```

# 3. DEQUE

В **очереди в двусторонним доступом** элементы имеют два указателя. Как и в обычной **очереди**, имеется доступ только к *tail* и *head*.

```
struct Node{
    int value; //getting value
    Node *next, *prev;//pointers to next and previous elements
};

struct deque{
    Node *head=nullptr;//initialization of the beginning
    //and ending of the deque
    Node *tail=nullptr;
    int count=0;
    void push_back(int num){
        Node* element=new Node; //adding memory for the structure
        element->value=num; //adding value
        count++;
        if(!head){  //if deque is empty
            head=element; //'cause there is only one element,
            tail=head; //it is both head and tail
        }
        else{
            element->prev=tail; //prev element according to
            //the added one is gonna be the tail
            tail->next=element;
            tail=element; //new element is a tail
        }
        cout<<"ok"<<endl;
    }
    void push_front(int num){
        Node *element=new Node;
        element->value=num;
        count++;
        if(!head){
            head=element;
            tail=head;
        }
        else{
            element->next=head; //next element after added is tail
            head->prev=element;  //before head is element
```

```
            head=element;  //element is a head
        }
        cout<<"ok"<<endl;
    }
    void pop_back(){
        if(count!=0){     //if deque is not empty
            cout<<tail->value<<endl;
            if(count>1){
                Node *element=tail; //using tail
                tail=tail->prev;
                tail->next=nullptr;
                delete element; //deleting previous tail
                count--;
            }
            else{  //if count=1
                head=tail=0;
                count--;
            }
        }
        else cout<<"error"<<endl;
    }
    void pop_front(){
        if(count!=0){
            cout<<head->value<<endl;
            if(head->next){           //if count>1
                Node *element=head; //using head
                head=head->next;
                head->prev=nullptr;
                delete element;
                count--;
            }
            else if(head==tail){
                head->next=nullptr;
                head=nullptr;
                delete head;
                count=0;
            }
        }
        else cout<<"error"<<endl;
    }
    void back(){
        if(count!=0)cout<<tail->value<<endl;
        else cout<<"error"<<endl;
    }
    void front(){
        if(count!=0)cout<<head->value<<endl;
        else cout<<"error"<<endl;
    }
    void size(){
        cout<<count<<endl;
```

```
    }
    void clear(){
        count=0;
        cout<<"ok"<<endl;
        while(head)
     {
          tail=head->next;
          delete head;
          head=tail;
     }
    }
    void exit(){
        cout<<"bye";
    }
}
...
```

# 4. FORWARD LIST and LIST

В **односвязных списках** и **двусвязных списках** добавление элемента происходит путем переписывания указателей.

В связи с подробной реализацией предыдущих СД, укажем лишь параметры односвязного списка:

**Параметры forward list:**
alloc - Аллокатор, используемый для всех выделений памяти в контейнере.
count - Размер контейнера.
value - Значение, которым будут инициализированы элементы контейнера.
first, last - Диапазон, из которого копируются элементы.
other - другой контейнер, который будет использоваться в качестве источника для инициализации элементов контейнера.
init - список инициализации элементов контейнера.

**Двусвязный список (list)** представляет собой контейнер, который поддерживает быструю вставку и удаление элементов из любой позиции в контейнере. Быстрый произвольный доступ не поддерживается. Он реализован в виде двусвязного списка. В отличие от std::`forward_list` (односвязного списка) этот контейнер обеспечивает возможность двунаправленного итерирования, являясь при этом менее эффективным в отношении используемой памяти.

# 5. PRIORITY QUEUE

**Очередь с приоритетом** - это тип контейнера, который позволяет достичь константной скорости доступа к максимальному (или минимальному) элементу, за счет увеличения скорости вставки элементов в контейнер до логарифмической. Работа с `priority_queue` похожа на работу с кучей в контейнерах случайного доступа, но имеет преимущество в

виде невозможности случайного повреждения кучи. Пример: диспетчеризация задач.

**Реализация с использованием linked list:**

```
// Node
typedef struct node {
    int data;

    // Lower values indicate higher priority
    int priority;

    struct node* next;

} Node;

// Function to Create A New Node
Node* newNode(int d, int p)
{
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = d;
    temp->priority = p;
    temp->next = NULL;
    return temp;
}

// Return the value at head
int peek(Node** head)
{
    return (*head)->data;
}

// Removes the element with the
// highest priority form the list
void pop(Node** head)
{
    Node* temp = *head;
    (*head) = (*head)->next;
    free(temp);
}

// Function to push according to priority
void push(Node** head, int d, int p)
{
    Node* start = (*head);

    // Create new Node
    Node* temp = newNode(d, p);

    // Special Case: The head of list has lesser
    // priority than new node. So insert new
```

```
    // node before head node and change head node.
    if ((*head)->priority < p) {

        // Insert New Node before head
        temp->next = *head;
        (*head) = temp;
    }
    else {

        // Traverse the list and find a
        // position to insert new node
        while (start->next != NULL &&
               start->next->priority < p) {
            start = start->next;
        }

        // Either at the ends of the list
        // or at required position
        temp->next = start->next;
        start->next = temp;
    }
}

// Function to check is list is empty
int isEmpty(Node** head)
{
    return (*head) == NULL;
}
...
```