**1. Динамические структуры данных, виды.**  
Динамические структуры данных – это структуры, элементы которых размещаются в памяти во время выполнения программы (на куче) и связаны указателями. К основным видам относятся:

* **Односвязные списки** – последовательность узлов, где каждый узел содержит данные и указатель на следующий элемент.
* **Двусвязные списки** – узлы имеют указатели как на следующий, так и на предыдущий элемент.
* **Деревья** (например, бинарные, AVL, красно-чёрные) – иерархическая структура, где каждый узел может иметь несколько потомков.
* **Графы** – набор вершин, связанных ребрами, где связи задаются с помощью списков смежности или матриц смежности.
* **Кучи (heap)** – специализированное дерево для организации приоритетной очереди.
* Иные структуры: хеш-таблицы, списки с произвольной связью и т.д.

**2. Размещение в памяти динамических величин. За счет каких ресурсов выделяется память под динамические структуры?**  
Память для динамических структур выделяется в **куче (heap)**, также называемой «свободным хранилищем». Выделение происходит с помощью операторов:

* **new / delete** (в C++),
* **malloc / free** (на уровне C).  
  Эта область памяти управляется операционной системой через библиотеку времени выполнения, что позволяет динамически запрашивать и освобождать память в ходе работы программы.

**3. Доступ к динамическим структурам из программного кода.**  
Доступ осуществляется посредством указателей:

* **Хранение адреса** начального элемента (например, головы списка или корня дерева).
* Использование операторов разыменования (\*) и обращения к членам через -> для доступа к данным и связям между узлами.  
  Пример для односвязного списка:

struct Node {

int data;

Node\* next;

};

void printList(Node\* head) {

while (head) {

std::cout << head->data << " ";

head = head->next;

}

}

**4. Смежное и связное представление данных в динамических структурах.**

* **Смежное представление** – данные хранятся в непрерывном участке памяти (например, массив или вектор). Это обеспечивает быстрый доступ по индексу, но вставка/удаление элементов может требовать сдвига остальных элементов.
* **Связное представление** – данные распределены по памяти, а связи между элементами задаются указателями. Вставка и удаление происходят быстрее, так как требуется лишь перенастройка указателей, но доступ по индексу менее эффективен (последовательный перебор).

**5. Какого типа может быть поле данных в динамической структуре?**  
Поле данных может быть практически **любого типа**:

* Примитивные типы (например, int, double),
* Пользовательские типы (структуры, классы),
* Указатели (на динамические структуры или другие данные),
* Даже объекты стандартных контейнеров STL.  
  Главное требование – тип должен корректно копироваться или перемещаться в зависимости от логики структуры.

**6. Односвязные списки. Операции добавления, удаления и поиска данных.**

* **Добавление:**  
  Вставка нового узла может производиться, например, в начало списка:
* void insertAtHead(Node\*& head, int value) {
* Node\* newNode = new Node{value, head};
* head = newNode;
* }
* **Удаление:**  
  Для удаления узла необходимо перенастроить указатель предыдущего узла и освободить память удаляемого:
* void deleteNode(Node\*& head, int value) {
* if (!head) return;
* if (head->data == value) {
* Node\* temp = head;
* head = head->next;
* delete temp;
* return;
* }
* Node\* current = head;
* while (current->next && current->next->data != value)
* current = current->next;
* if (current->next) {
* Node\* temp = current->next;
* current->next = temp->next;
* delete temp;
* }
* }
* **Поиск:**  
  Перебор списка с сравнением значения каждого узла:
* Node\* search(Node\* head, int value) {
* while (head) {
* if (head->data == value)
* return head;
* head = head->next;
* }
* return nullptr;
* }

**7. Двусвязные списки. Операции добавления, удаления и поиска данных.**  
Структура узла содержит два указателя – на следующий и предыдущий элемент.

* **Добавление:**  
  Пример добавления нового узла после заданного:
* struct DNode {
* int data;
* DNode\* next;
* DNode\* prev;
* };
* void insertAfter(DNode\* node, int value) {
* if (!node) return;
* DNode\* newNode = new DNode{value, node->next, node};
* if (node->next)
* node->next->prev = newNode;
* node->next = newNode;
* }
* **Удаление:**  
  Для удаления узла необходимо перенастроить оба указателя соседних узлов:
* void deleteNode(DNode\*& head, DNode\* node) {
* if (!node) return;
* if (node->prev)
* node->prev->next = node->next;
* else
* head = node->next; // удаляется первый элемент
* if (node->next)
* node->next->prev = node->prev;
* delete node;
* }
* **Поиск:**  
  Аналогичен поиску в односвязном списке, с перебором и сравнением значения каждого узла.

**8. В чем отличие первого элемента однонаправленного (двунаправленного) списка от остальных элементов этого же списка?**

* **Односвязный список:**  
  Первый элемент (голова) является отправной точкой для доступа ко всему списку, при этом он не имеет «предыдущего» узла (понятие предыдущего не применяется).
* **Двусвязный список:**  
  Первый элемент имеет указатель prev, который **обычно равен nullptr**, что позволяет отличить его от остальных узлов, у которых этот указатель указывает на предыдущий элемент.

**9. В чем отличие последнего элемента однонаправленного (двунаправленного) списка от остальных элементов этого же списка?**

* **Односвязный список:**  
  Последний элемент характеризуется тем, что его указатель next равен nullptr, что сигнализирует об окончании списка.
* **Двусвязный список:**  
  Аналогично, последний элемент имеет указатель next, равный nullptr, что отличает его от остальных, у которых этот указатель указывает на следующий узел.

**10. Стек и очередь. Операции добавления, удаления и поиска данных.**

* **Стек (LIFO – Last In First Out):**
  + **Добавление (push):** помещает элемент на вершину стека.
  + **Удаление (pop):** удаляет элемент с вершины стека.
  + **Доступ (top):** возвращает значение элемента на вершине, не удаляя его.  
    Пример:
* std::stack<int> s;
* s.push(10);
* int topElem = s.top();
* s.pop();
* **Очередь (FIFO – First In First Out):**
  + **Добавление (push/enqueue):** добавляет элемент в конец очереди.
  + **Удаление (pop/dequeue):** удаляет элемент из начала очереди.
  + **Доступ (front/back):** возвращает элементы в начале или конце очереди.  
    Пример:
* std::queue<int> q;
* q.push(20);
* int firstElem = q.front();
* q.pop();

*Примечание:* В стандартных контейнерах поиск не является основной операцией для стеков и очередей – при необходимости можно перебрать элементы через доступ к базовому контейнеру (если он доступен).

**11. С какой целью в программах выполняется удаление однонаправленного (двунаправленного) списка по окончании работы с ним? Как изменится работа программы, если операцию удаления списка не выполнять?**  
Удаление динамически выделенной памяти (то есть, всех узлов списка) необходимо для:

* **Освобождения ресурсов памяти,** предотвращения утечек памяти (memory leaks), которые при длительной работе или в долгоживущих программах могут привести к исчерпанию памяти.
* **Поддержания стабильности и корректной работы программы.**  
  Если не выполнять удаление, программа будет постоянно резервировать память, что может привести к снижению производительности и даже аварийному завершению работы при исчерпании доступной памяти.

**12. Контейнеры в STL, виды контейнеров.**  
STL предоставляет широкий набор контейнеров, которые можно разделить на следующие группы:

* **Последовательные контейнеры:**
  + vector – динамический массив, обеспечивающий быстрый доступ по индексу.
  + list – двусвязный список.
  + deque – двусторонняя очередь, позволяющая эффективно работать с обоими концами.
  + forward\_list – односвязный список (в версиях C++11 и выше).
  + array – фиксированный по размеру массив.
* **Ассоциативные контейнеры:**
  + set, multiset, map, multimap – элементы упорядочены с использованием бинарных деревьев.
* **Контейнеры с хешированием (unordered):**
  + unordered\_set, unordered\_multiset, unordered\_map, unordered\_multimap – обеспечивают быстрый доступ за счёт хеширования.
* **Адаптеры контейнеров:**
  + stack, queue, priority\_queue – реализованы поверх вышеупомянутых контейнеров и предоставляют специализированный интерфейс.

**13. Понятие итератора в библиотеке STL. Операции над итераторами.**  
Итератор – это абстракция, похожая на указатель, которая обеспечивает доступ и перемещение по элементам контейнера. Основные операции:

* **Инкремент (operator++):** переход к следующему элементу.
* **Декремент (operator--):** для двунаправленных итераторов – переход к предыдущему элементу.
* *Разыменование (operator и operator->):*\* доступ к значению элемента.
* **Операции сравнения (==, !=):** для проверки равенства итераторов.  
  Для контейнеров с произвольным доступом (например, vector) итераторы поддерживают арифметические операции (например, +, -, оператор индексирования []).

**14. Контейнер list в библиотеке STL. Основные методы.**  
Контейнер std::list представляет двусвязный список. Основные методы:

* **Модификаторы:**
  + push\_front() и push\_back() – добавление элементов в начало или конец.
  + pop\_front() и pop\_back() – удаление элементов с начала или конца.
  + insert() – вставка элементов в указанное место.
  + erase() – удаление элементов по итератору или диапазону.
  + clear() – удаление всех элементов.
* **Наблюдатели:**
  + front() и back() – доступ к первому и последнему элементу.
* **Операции с итераторами:**
  + begin() и end() – получение итераторов для прохода по списку.
* **Другие методы:**
  + size(), empty().

**15. Итераторы для list. Допустимые операции над итераторами.**  
Итераторы std::list являются **двунаправленными**:

* Допустимы операции **инкремента** (++it) и **декремента** (--it).
* Поддерживается операция **разыменования** (\*it).
* **Сравнение итераторов** (например, it == anotherIt).
* **Арифметические операции** (сложение/вычитание чисел) недоступны, так как список не обеспечивает произвольного доступа.

**16. Контейнер stack в библиотеке STL. Основные методы.**  
std::stack – это адаптер контейнера, реализующий принцип LIFO. Основные методы:

* push() – добавление элемента на вершину стека.
* pop() – удаление элемента с вершины.
* top() – доступ к элементу на вершине без удаления.
* empty() – проверка на пустоту.
* size() – получение количества элементов.  
  *Замечание:* Реализация стека по умолчанию использует контейнер deque, но можно использовать и другой последовательный контейнер.

**17. Контейнер vector в библиотеке STL. Основные методы.**  
std::vector – динамический массив, который автоматически изменяет размер. Основные методы:

* push\_back() – добавление элемента в конец.
* pop\_back() – удаление последнего элемента.
* insert() – вставка элемента/диапазона элементов в произвольное место.
* erase() – удаление элемента/диапазона элементов.
* operator[] и at() – доступ к элементам по индексу.
* size(), capacity(), resize(), empty().
* begin() и end() – для получения итераторов.

**18. Итераторы для vector. Допустимые операции над итераторами.**  
Итераторы std::vector являются **итераторами произвольного доступа**:

* Поддерживают **инкремент** (++it) и **декремент** (--it).
* Можно выполнять **арифметические операции**: прибавлять/вычитать целые числа (например, it + 3, it - 2).
* Поддерживается операция **вычитания** двух итераторов (для определения расстояния).
* Допустимы операции **сравнения** (например, <, >, <=, >=).
* Можно использовать оператор индексирования (it[n]) для доступа к элементу, находящемуся на расстоянии n от текущей позиции.