Динамические структуры данных. Связные списки (однонаправленный и двунаправленный, циклический). Основные операции, их вычислительная сложность

Динамическая структура данных характеризуется тем что:

она не имеет имени;

ей выделяется память в процессе выполнения программы;

количество элементов структуры может не фиксироваться;

размерность структуры может меняться в процессе выполнения программы;

в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Каждой динамической структуре данных сопоставляется статическая переменная типа указатель (ее значение – адрес этого объекта), посредством которой осуществляется доступ к динамической структуре.

Динамические структуры, по определению, характеризуются отсутствием физической смежности элементов структуры в памяти, непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки.

Классификация динамических структур данных:

Они отличаются способом связи отдельных элементов и/или допустимыми операциями. Динамическая структура может занимать несмежные участки оперативной памяти.

Под структурами данных подразумевается хранение данных и их организация таким образом, чтобы решать поставленную задачу наиболее эффективным способом. В Java есть следующие структуры данных:

- Массив
- Список (Динамический массив)
- Стек
- Очередь
- Связный список
- HashTable и HashMap
- Дерево

Однонаправленные связные списки:

- Операции:
  - Вставка элемента в начало списка (push) - O(1)
  - Удаление элемента из начала списка (pop) - O(1)
  - Вставка элемента после определенного элемента (insert) - O(1)

        - Удаление определенного элемента (remove) - O(n)
    - Пример реализации:

```
class Node {
    int data;
    Node next;

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

class LinkedList {
    Node head;

    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }

    public void pop() {
        if (head == null) {
            return;
        }
        head = head.next;
    }

    public void insert(int data, Node prevNode) {
        if (prevNode == null) {
            return;
        }
        Node newNode = new Node(data);
        newNode.next = prevNode.next;
        prevNode.next = newNode;
    }

    public void remove(int data) {
        Node temp = head;
        Node 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;
    }
}
```

Двунаправленные связные списки:

- Операции:
  - Вставка элемента в начало списка (push) - O(1)
  - Удаление элемента из начала списка (pop) - O(1)
  - Вставка элемента после определенного элемента (insert) - O(1)
  - Удаление определенного элемента (remove) - O(n)
- Пример реализации:

javaCopy code

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

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

class DoublyLinkedList {
    Node head;

    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        if (head != null) {
            head.prev = newNode;
        }
        head = newNode;
    }

    public void pop() {
        if (head == null) {
```

```
            return;
        }
        head = head.next;
        head.prev = null;
    }

    public void insert(int data, Node prevNode) {
        if (prevNode == null) {
            return;
        }
        Node newNode = new Node(data);
        newNode.next = prevNode.next;
        newNode.prev = prevNode;
        if (prevNode.next != null) {
            prevNode.next.prev = newNode;
        }
        prevNode.next = newNode;
    }

    public void remove(int data) {
        if (head == null) {
            return;
        }
        if (head.data == data) {
            head = head.next;
            head.prev = null;
            return;
        }
        Node temp = head;
        while (temp != null && temp.data != data) {
            temp = temp.next;
        }
        if (temp == null) {
            return;
        }
        if (temp.next != null) {
            temp.next.prev = temp.prev;
        }
        temp.prev.next = temp.next;
    }
}
```

Циклический связный список в Java можно реализовать с помощью класса, который представляет узел списка и содержит ссылку на следующий узел. Вот пример реализации:

```
public class ListNode<T> {
    private T value;
    private ListNode<T> next;

    public ListNode(T value) {
        this.value = value;
        this.next = null;
    }

    public T getValue() {
        return value;
    }

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

    public ListNode<T> getNext() {
        return next;
    }

    public void setNext(ListNode<T> next) {
        this.next = next;
    }
}

public class CircularLinkedList<T> {
    private ListNode<T> tail;

    public CircularLinkedList() {
        tail = null;
    }

    public boolean isEmpty() {
        return tail == null;
    }

    public void insertAtBeginning(T value) {
        ListNode<T> newNode = new ListNode<>(value);

        if (isEmpty()) {
            newNode.setNext(newNode);
            tail = newNode;
        } else {
            newNode.setNext(tail.getNext());
            tail.setNext(newNode);
        }
```

```
    }

    public void insertAtEnd(T value) {
        insertAtBeginning(value);
        tail = tail.getNext();
    }

    public void delete(T value) {
        if (!isEmpty()) {
            ListNode<T> current = tail.getNext();

            if (current != tail && current.getValue().equals(value)) {
                tail.setNext(current.getNext());
            } else {
                while (current != tail && !current.getNext().getValue().equals(value)) {
                    current = current.getNext();
                }

                if (current != tail) {
                    current.setNext(current.getNext().getNext());
                    if (current.getNext() == tail) {
                        tail = current;
                    }
                }
            }
        }
    }

    public void display() {
        if (!isEmpty()) {
            ListNode<T> current = tail.getNext();

            while (current != tail) {
                System.out.print(current.getValue() + " ");
                current = current.getNext();
            }

            System.out.print(current.getValue());
        }
    }
}

public class Main {
    public static void main(String[] args) {
        CircularLinkedList<Integer> list = new CircularLinkedList<>();
        list.insertAtBeginning(1);
        list.insertAtEnd(2);
```

```
        list.insertAtEnd(3);
        list.display(); // Output: 1 2 3
        list.delete(2);
        list.display(); // Output: 1 3
    }
}
```

Массив

Массив - это нумерованный набор переменных одного типа.

Объявляется следующем образом:
`int[] arr = new int[10];`

- Все массивы в Java одномерные. В случае с многомерными массивами каждый элемент содержит только ссылку на вложенный массив
- Можно создать нулевого размера, может быть полезно если нужно вернуть пустой массив из какого-либо метода
- Оператор new используется для создания ссылочного типа данных. Ссылка хранится на стеке, а объект в куче. Если на объект нет ссылок, то он будет удалён автоматически. Удаление объекта может быть осуществлено с задержкой

Список (Динамический массив)

Идея списка или же динамического массива заключается в автоматическом расширении емкости.

Объявляется следующем образом:
`ArrayList<Integer> arr = new ArrayList<Integer>();`

- Примитивный тип данных передать не можем, поэтому передаем класс обертку. О классах обертках, можно прочитать здесь. При желании можно написать универсальную реализацию ArrayList, сделать его массивом Object и тогда можно будет хранить еще и примитивы благодаря автоупаковке
- Если не указать в конструктор начальную емкость, то будет создан пустой список с емкостью в 10 элементов
- В случае, когда зарезервированной емкости не хватает, при достижении максимального количества элементов будет создан новый массив с емкостью: новая_емкость = (старая_емкость * 3) / 2 + 1. Существующие элементы списка будут скопированы в новый массив
- Чтобы не тратить память напрасно, при удалении элементов следует вызывать метод trimToSize()

Стек

Очередь работает по принципу LIFO. В Java наследуется от Vector<E>, реализует следующие интерфейсы: Iterable<E>, Collection<E>, List<E>, RandomAccess, Serializable, Cloneable.

Объявляется следующем образом:
Stack<Integer> arr = new Stack<Integer>();

- push() - добавляет в конец очереди;
- peek() - возвращает последний элемент и *не удаляет* его;
- pop() - *удаляет* последний элемент и возвращает его;
- empty() - вернет true - если очередь пуста и false - в противном случае;
- search() - возвращает *номер позиции* с конца очереди.

Очередь

Интерфейс Queue<E> описывает одностороннюю очередь, а Deque<E> - двухстороннюю. Прежде чем перейти к объявлению в Java, стоит отметить иерархию наследования. Иерархия следующая:

- Iterable<T> => Collection<E> => Queue<E> => Deque<E>

Интерфейсы Queue<E> и Deque<E> реализуют следующие классы:

- ArrayDeque<E> - двухсторонняя очередь
- LinkedList<E> - связный список
- PriorityQueue<E> - очередь с приоритетами

Объявляется следующем образом:
Queue<Integer> arr = new ArrayDeque<Integer>();
Deque<Integer> arr1 = new ArrayDeque<Integer>();
PriorityQueue<Integer> arr2 = new PriorityQueue<Integer>();
// Очередь на LinkedList'e
Queue<Integer> arr = new LinkedList<Integer>();
Deque<Integer> arr = new LinkedList<Integer>();

- ArrayDeque реализует дек на массиве, поэтому он эффективнее по памяти и работает быстрее, чем LinkedList

Пару слов о PriorityQueue.
Этот класс реализует следующие интерфейсы: Iterable<E>, Collection<E>, Queue<E>, Serializable. У этого класса есть свои особенности:

- Из очереди первым возвращается элемент с наибольшим приоритетом
- Значение null добавить нельзя

Связный список

LinkedList<E> реализует связный список, элементы которого хранят ссылки на предыдущий и следующий элементы.

Класс реализует следующие интерфейсы:
Iterable<E>, Collection<E>, List<E>, Queue<E>, Deque<E>, Serializable, Cloneable.

Объявляется следующем образом:
LinkedList<Integer> arr = new LinkedList<Integer>();

Операция	ArrayList	LinkedList
add (E element)	O(1) O(n) - при копировании	O(1)
add (int index, E element)	O(n/2) - с середины O(n) - с начала O(1) - с конца	O(n/4) O(n) - в конец или начало
remove (int index)	O(n/2) - с середины O(n) - с начала O(1) - с конца	O(n/4) O(n) - в конец или начало
get (int index)	O(1)	O(n/4)

LinkedList занимает гораздо больше памяти, чем ArrayList. Использовать нужно в определенных случаях, чаще всего когда речь идет о двусвязном списке. Также стоит отметить, что элементы у ArrayList в памяти хранятся линейно, поэтому доступ по индексу происходит за O(1)

HashTable и HashMap

HashTable считается устаревшей, поэтому приведена лишь разница между мапой и таблицей. HashMap используется для хранения пары «ключ-значение». В качестве примера использования хэш-мапы можно привести пациента больницы, у которого есть Ф.И.О. и номер медицинского полиса.

- Если конструктору не передать никаких значений, то будет создан пустой словарь с емкостью в 16 элементов и коэффициентом заполнения 0.75
- Если коэффициент заполнения достигает максимума, то число bucket'ов увеличивается в два раза

Класс HashMap<K, V> реализует следующие интерфейсы:
Map<K, V>, Serializable, Cloneable.

Объявляется следующем образом:
HashMap<String, Integer> map = new HashMap<String, Integer>();

- Хэш-Таблица не может хранить null, в отличии от Хэш-Мапы
- В Хэш-Таблице все методы синхронизированы, что сказывается на скорости работы
- Хэш-Таблица не рекомендуется к использования, так как считается устаревшей, Хэш-Мапа предпочтительнее

P.S. Если требуется выбрать структуру, которая справится с параллельными вычислениями, то есть ConcurrentHashMap

Дерево

Стоит заметить, что готовой реализации бинарного дерева в Java нет, но есть TreeMap<K, V> и TreeSet<E>, которые описывают словари, где ключи хранятся в отсортированном порядке. TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов.

Класс TreeSet<E> реализует следующие интерфейсы:
Itearble<E>, Collection<E>, Set<E>, SortedSet<E>, NavigatbleSet<E>, Serializable, Cloneable.

Класс TreeMap<K, V> реализует следующие интерфейсы:
Map<K,V>, SortedMap<K, V>, NavigatbleMap<K, V>, Serializable, Cloneable.

TreeSet<Integer> set = new TreeSet<Integer>(); TreeMap<String, Integer> map = new TreeMap<String, Integer>();

Связные списки являются динамическими структурами данных, которые позволяют хранить и организовывать коллекцию элементов. В связном списке каждый элемент, называемый узлом, содержит значение данных и ссылку на следующий элемент в списке. Вот основные операции, выполняемые над связными списками, и их вычислительная сложность:

Вставка в начало списка (однонаправленный/двунаправленный): Операция вставки элемента в начало списка требует только изменения ссылок на следующий элемент (и предыдущий элемент для двунаправленного списка) и, следовательно, выполняется за время O(1).

Вставка в конец списка (однонаправленный/двунаправленный): При вставке элемента в конец списка необходимо пройти по всему списку до конца для обновления ссылки последнего элемента на новый узел. Таким образом, операция выполняется за время O(n), где n - это размер списка.

Вставка в произвольную позицию (однонаправленный/двунаправленный): Для вставки элемента в произвольную позицию списка необходимо найти эту позицию, обновить ссылки предыдущего и текущего элемента, и вставить новый элемент. Эта операция требует времени O(n), где n - это позиция, в которую вставляется элемент.

Удаление из начала списка (однонаправленный/двунаправленный): При удалении элемента из начала списка, мы просто обновляем ссылку на первый элемент. Эта операция выполняется за время O(1).

Удаление из конца списка (однонаправленный/двунаправленный): Удаление элемента из конца списка требует пройти по списку до предпоследнего элемента, чтобы

обновить ссылку на следующий элемент и удалить последний элемент. Таким образом, операция выполняется за время O(n), где n - это размер списка.

Удаление из произвольной позиции (однонаправленный/двунаправленный): Удаление элемента из произвольной позиции требует поиска этой позиции, обновления ссылок предыдущего и следующего элементов, и удаления элемента. Эта операция требует времени O(n), где n - это позиция, из которой удаляется элемент.

Поиск элемента в списке: Поиск элемента в связном списке требует пройти по всему списку, пока не будет найден нужный элемент, или пока не будут просмотрены все элементы, в случае отсутствия искомого элемента в списке. Эта операция выполняется в среднем за время O(n), где n - это размер списка.

Важно отметить, что для двунаправленных связных списков операции вставки и удаления могут потребовать дополнительных операций по обновлению ссылок на предыдущие элементы. Тем не менее, общие сложности указанных операций справедливы для обоих типов связных списков.