Языки программирования

Лекция 13

ПМИ 2 курс

Демяненко Я.М.

ЮФУ 2024

Абстрактный тип данных (АТД)

- Это тип данных,
  - который предоставляет для работы с элементами этого типа определённый набор функций,
  - а также возможность создавать элементы этого типа при помощи специальных функций.

- Суть абстракции
  - Вся внутренняя структура спрятана от разработчика программного обеспечения.
  - Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями.

- Конкретные реализации АТД называются структурами данных.

Интерфейсы

- В программировании АТД обычно представляются в виде интерфейсов, которые скрывают соответствующие реализации типов.

- Программисты работают с АТД исключительно через их интерфейсы, поскольку реализация может в будущем измениться.

Инкапсуляция

- Такой подход соответствует принципу инкапсуляции в объектно-ориентированном программировании.
- Пока структура данных поддерживает интерфейс, все программы, работающие с заданной структурой АТД, будут продолжать работать.
- Разработчики структур данных стараются, не меняя внешнего интерфейса и семантики функций, постепенно дорабатывать реализации, улучшая алгоритмы по скорости, надежности и используемой памяти.

Модульность

- АТД позволяют достичь модульности программных продуктов и иметь несколько альтернативных взаимозаменяемых реализаций отдельного модуля.

Примеры АТД

- Список
- Стек
- Очередь
- Дек
- Очередь с приоритетом
- Ассоциативный массив

Список

- Это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза.
- Экземпляры значений, находящихся в списке, называются элементами списка; если значение встречается несколько раз, каждое вхождение считается отдельным элементом.
- Термином список также называется несколько конкретных структур данных, применяющихся при реализации абстрактных списков, особенно связных списков

Списки в функциональных языках

- Являются фундаментальной структурой.

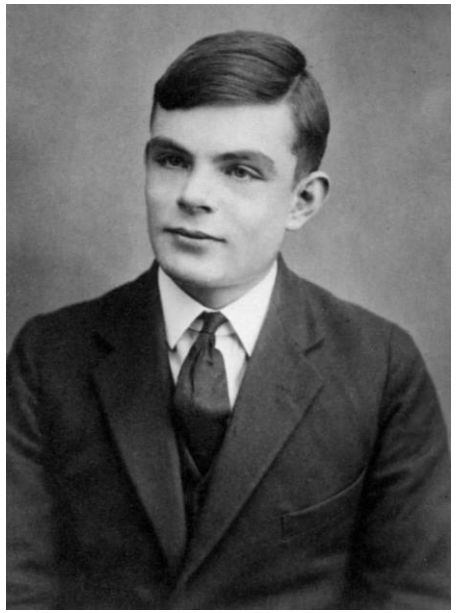
- Большинство функциональных языков имеет встроенные средства для работы со списками:
  - получение длины списка,
  - головы (первый элемент списка),
  - хвоста (часть списка, идущая за первым элементом),
  - применения функции к каждому элементу списка (Map),
  - свертки списка и пр.

- Haskell
- Lisp

Стек

- Структура данных, представляющая собой список элементов, организованных по принципу **LIFO** (англ. last in — first out, «последним пришёл — первым вышел»)
- **Стек задается своей вершиной**
- Добавление в вершину, извлечение из вершины

- В 1946 Алан Тьюринг ввёл понятие стека.



А́лан Мэ́тисон Тью́ринг — английский математик, логик, криптограф, оказавший существенное влияние на развитие информатики.
(1912-1954)

- А в 1957 году немцы Клаус Самельсон и Фридрих Л. Бауэр запатентовали идею.

Клаус Самельсон, один из создателей языка ALGOL
(1918-1980)

Фридрих Л. Бауэр,
1924-2015

Система команд стека

- push Добавить в стек новый элемент
- pop Извлечь из стека элемент
- top Узнать значение элемента в вершине стека (не удаляя его)
- clear Очистить стек (удалить из него все элементы)
- isEmpty Проверить на пустоту

Стек на базе массива фиксированного размера

```
#define SIZE 50
int *top, *p1, stack[SIZE];

 void init() {
 top = stack; p1 = stack;
}

void push(int x) {
  if (p1 == (top+SIZE)) {
      throw("Stack Overflow"); //exit(1);
  }
  *p1 = x;
  p1++;
}
```

```
int pop() {
  if(p1 == top) {
     throw("Stack Underflow"); //exit(1);
  }
  p1--;
  return *p1;
}
```

Стек на базе массива переменного размера

```
m_size = 0; m_maxsize = 1; m_data = new T[m_maxsize];

void resize(int newsize) {
  T * newdata = new T[newsize];
  for (int i = 0; i < m_size; ++i)
       newdata[i] = m_data[i];
  delete[] m_data; m_data = newdata; m_maxsize = newsize;
}

void push(T val) {
  if (m_size == m_maxsize)
     resize(2 * m_maxsize);
  m_data[m_size] = val;
  ++m_size;
}
```

Очередь

- Структура данных с доступом к элементам по принципу «первый пришёл — первый вышел» (**FIFO**, First In — First Out).
- **Очередь задается своей головой (началом) и хвостом (концом)**
- Добавление элемента (принято обозначать словом enqueue) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть словом dequeue), при этом выбранный элемент из очереди удаляется.

Система команд очереди

- push (enqueue) — Добавить в очередь (в хвост) новый элемент
- pop (dequeue) — Извлечь из очереди (из головы) элемент
- top — Узнать значение элемента в голове очереди (не удаляя его)
- clear — Очистить очередь (удалить из неё все элементы)
- isEmpty — Проверить на пустоту

Очередь на базе массива

```
const int N=4; //размер очереди
struct Queue {
        int data[N]; //массив данных
        int last; //номер хвостового элемента
};

void Creation (Queue *Q) {Q->last=0; }

bool isEmpty (Queue *Q) {
  return Q->last==0;
}
```

```
void Enqueue(Queue *Q, int value) {
  if (Q->last==N) {
     throw("Overflow");
  }
  Q->data[Q->last++]=value;
}

void Dequeue(Queue *Q) {
  for (int i=0; i<Q->last; i++)  //смещение элементов
       Q->data[i]=Q->data[i+1];
  Q->last--;
}

int Top(Queue *Q) {
   return Q->data[0];
}

int Size(Queue *Q)  {
    return Q->last;
}
```

Дек (двусвязная очередь)

- (от англ. deque — double ended queue; двухсторонняя очередь, двусвязный список, очередь с двумя концами) — структура данных, в которой элементы можно добавлять и удалять как в начало, так и в конец, то есть **одновременно FIFO и LIF**O

- Система команд дека:
  - push_front — Добавить (положить) в начало дека новый элемент
  - push_back — Добавить (положить) в конец дека новый элемент
  - pop_front — Извлечь из дека первый элемент
  - pop_back — Извлечь из дека последний элемент
  - front — Узнать значение первого элемента (не удаляя его)
  - back — Узнать значение последнего элемента (не удаляя его)
  - size — Узнать количество элементов в деке
  - clear — Очистить дек (удалить из него все элементы)

Очередь с приоритетом

- АТД, поддерживающий три операции:

- InsertWithPriority — добавить в очередь элемент с назначенным приоритетом
- GetNext — извлечь из очереди и вернуть элемент с наивысшим приоритетом.
- PeekAtNext (необязательная операция) — просмотреть элемент с наивысшим приоритетом без извлечения

- Позволяет хранить пары (ключ, значение) и поддерживает операции добавления пары, поиска пары с минимальным ключом и извлечения пары с минимальным ключом:

- INSERT(ключ, значение) — добавляет пару в хранилище;
- MIN — возвращает пару с минимальным значением ключа.
- EXTRACT_MIN — возвращает пару с минимальным значением ключа, удаляя её из хранилища.

- Очередь с приоритетом может хранить несколько пар с одинаковыми ключами.

Расширения очереди с приоритетом

Различные реализации очереди с приоритетом нередко расширяют её интерфейс следующими операциями:

- DELETE(k) — удалить пару с ключом k;
- CHANGE_KEY(k, k_new) — в первой паре с ключом k заменить ключ на k_new;
- UNION(queue1, queue2) — из двух очередей с приоритетом сделать одну, объединив множества хранимых в них пар.

Множество

Тип данных, хранящий информацию о присутствии в множестве объектов любого счетного типа

Ассоциативный массив (словарь)

- АТД, позволяющий хранить пары вида «(ключ, значение)»
- поддерживающий операции добавления пары, а также поиска и удаления пары по ключу:
  - INSERT(ключ, значение)
  - FIND(ключ)
  - REMOVE(ключ)

Расширения ассоциативного массива

- CLEAR — удалить все записи
- EACH — «пробежаться» по всем хранимым парам
- MIN — найти пару с минимальным значением ключа
- MAX — найти пару с максимальным значением ключа

В последних двух случаях необходимо, чтобы на ключах была определена **операция сравнения**.

Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов — например, строки.

Ассоциативный массив не может хранить две пары с одинаковыми ключами

Реализации ассоциативного массива

- Самая простая реализация может быть основана на обычном массиве, элементами которого являются **пары (ключ, значение)**.
- Для ускорения операции поиска можно упорядочить элементы этого массива по ключу и осуществлять нахождение методом бинарного поиска.
- Но это увеличит время выполнения операции добавления новой пары, так как необходимо будет «раздвигать» элементы массива, чтобы в образовавшуюся пустую ячейку поместить новую запись.

- Наиболее популярны **реализации**, основанные **на различных деревьях поиска**.
- Так, например, в стандартной библиотеке STL языка C++ контейнер map реализован на основе красно-чёрного дерева.
- В языках Ruby, Tcl, Python используется один из вариантов хэш-таблицы.
- Есть и другие реализации

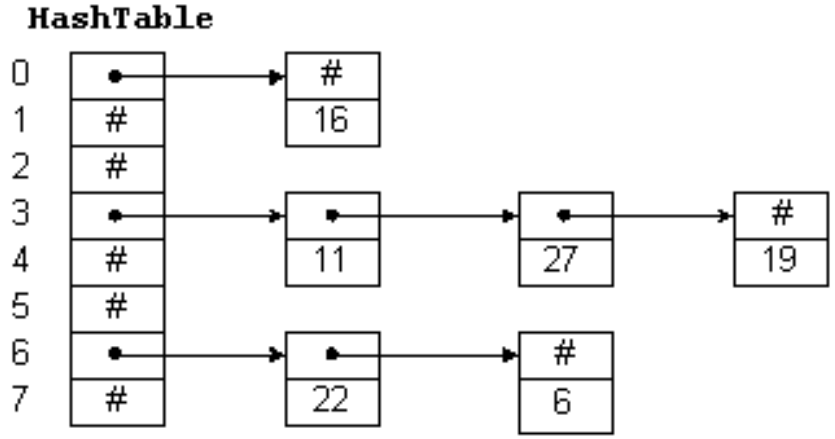
- У каждой реализации есть свои достоинства и недостатки.
- **Важно**, чтобы **все три операции** выполнялись как в среднем, так и в худшем случае за время **O(log n)** , где n — текущее количество хранимых пар.
- Для сбалансированных деревьев поиска (в том числе для красно-чёрных деревьев) это условие выполнено.

Требования к хеш-таблице

- Должна быть быстрой
- Должна порождать хорошие ключи для распределения элементов по таблице
- Последнее требование минимизирует коллизии (случаи, когда два разных элемента имеют одинаковое значение хеш-функции) и предотвращает случай, когда элементы данных с близкими значениями попадают только в одну часть таблицы.

Хеш-таблица

- Это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу
- Изобретено в 1953 году
- Изложение хеширования можно найти в работах Кормена[1990] и Кнута[1998]



Практическое применение

- Хеширование полезно, когда широкий диапазон возможных значений должен быть сохранен в малом объеме памяти, и нужен способ быстрого, практически произвольного доступа.
- Хэш-таблицы часто применяются в базах данных, и, особенно, в языковых процессорах типа компиляторов и ассемблеров, где они изящно обслуживают таблицы идентификаторов.

- Таблица символов компилятора. Хеширование является методом ускорения поиска. Компилятор встречает некоторую лексему и пытается найти ее в своей базе данных или таблице символических имен. Таблица символических имен современного компилятора в MS Windows может содержать несколько тысяч или десятков тысяч лексем. Для ускорения поиска был придуман следующий прием. Компилятор, найдя лексему в тексте программы, определяет ее хеш-ключ. Наша лексема — это просто слово, состоящее из последовательности кодов символов. Для определения хеш-кода следует сложить все коды символов. Лексем с таким хеш-кодом в базе данных уже значительно меньше. Компилятор определяет номер списка с заданным кодом и перебирает этот список, а не всю базу данных.

- Для поиска информации о водителе лишь по его номеру в водительском удостоверении.
- Для баз данных телефонных номеров.
- Каталог книг.
- Для хранения паролей пользователей.
- Браузер хранит адреса посещенных страниц в хеш-таблице.