Лекция 20. Расширенное представление данных.

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Содержание

- Функции: дополнительные сведения о функции m alloc ()
- Использование Си для представления разнообразных типов данных
- Новые алгоритмы и увеличение возможностей концептуальной разработки программ
- Абстрактные типы данных

Представление данных(1)

В зависимости от поставленной задачи, не всегда массивы и массивы структур могут быть *эффективным* **представлением данных.** Для *эффективного* программирования требуется учитывать специфику задач и знать базовые структуры данных.

Представление данных (2)

Вектор — динамический массив, поддерживающий эффективное добавление и удаление элементов с конца.

Преимущества

- Работа с последовательными данными
- Простота реализации
- Быстрый доступ O(1)
- Динамическое расширение

Недостатки

- Сдвиги при удалении и добавлении
- Резервирование

Представление данных (3)

Очередь — структура FIFO (First In, First Out), где элементы добавляются в конец и удаляются из начала.

Преимущества

- Простота реализации
- Эффективность добавления и удаления O(1)
- Подходит для задач, где порядок имеет важное значение

Недостатки

- Ограниченная функциональность
- Расходы на хранение указателей

Представление данных (4)

Словарь — ассоциативный массив, хранящий пары ключ-значение для быстрого доступа по ключу.

Преимущества

- Удобство за счёт ассоциации
- Быстрый доступ по ключу O(1)
- Возможность хранить различные типы данных

Недостатки

- Высокая сложность реализации
- Реализация хэш-таблиц, индексов

Представление данных (5)

Список — связная структура данных, позволяющая эффективно добавлять/удалять элементы в начале, конце или произвольной позиции.

Преимущества

- Гибкость в управлении памятью
- Эффективное удаление/добавление O(1)

Недостатки

- Медленны доступ к элементам по индексу O(n)
- Расходы на хранение указателей

Представление данных (6)

- **Вектор** подходит для работы с последовательными данными, где важен быстрый доступ по индексу.
- **Очередь** идеальна для задач, где требуется строгий порядок обработки элементов.
- **Словарь** незаменим при работе с ассоциативными данными и быстрым доступом по ключу.
- **Список** предоставляет максимальную гибкость в управлении данными, особенно при частых вставках/удалениях.

Представление данных: словарь (1)

```
// Структура словаря
typedef struct {
    tdPair* head; // Начало списка
    int size;     // Размер словаря
} tdDict;
```

Представление данных: словарь (2)

```
// Структура пара (ключ-значение)
typedef struct tdPair {
  void* key;          // Ключ
  void* value;        // Значение
  struct tdPair* next; //Следующий узел
} tdPair;
```

Представление данных: словарь (3)

```
// Создание пустого словаря
tdDict* dict_create() {
 tdDict* d=(tdDict*)malloc(sizeof(tdDict));
 d->head = NULL;
 d->size = 0;
 return d;
}
```

Представление данных: словарь (4)

```
// Освобождение памяти
void dict_free(tdDict* d) {
   tdPair* curr = d->head;
   while(curr) {
      tdPair* temp = curr;
     curr = curr >next;  free(temp);
   }  free(d);
}
```

Представление данных: словарь (5)

```
// Демонстрационная программа
int main()
{
    tdDict* d = dict_create();
     // Работа с данными
    dict_free(d);
    return EXIT_SUCCESS;
}
```

Представление данных: словарь (6)

```
// Добавление/обновление пары ключ-значение
void dict_put(tdDict* d, void* k, void* v) {
  tdPair* curr = d->head;
  while (curr) {//Пров. на существ. ключа
    if (curr->key == k ||
    !strcmp((char*) curr->key, (char*)k)) {
    curr->value = v; return; }
    curr = curr->next; }
```

Представление данных: словарь (7)

```
tdPair* new_pair =
    (tdPair*)malloc(sizeof(tdPair));
new_pair->key = k; new_pair->value = v;
new_pair->next = d->head;
d->head = new_pair; d->size++;
}
```

Представление данных: словарь (8)

```
// Получение значения по ключу
void* dict_get(tdDict* d, void* k) {
 tdPair* current = d->head;
 while (current) {
 if (current->key == k ||
    !strcmp((char*)current->key, (char*)k)) {
    return current->value; }
        current = current->next; }
    return NULL; }
```

Представление данных: словарь (9)

```
// Проверка наличия ключа
bool dict_cont_key(tdDict* d, void* k) {
 tdPair* current = d->head;
 while (current) {
    if (current->key == k ||
        !strcmp((char*)current->key, (char*)k))
      return true;
 current = current->next; }
 return false; }
```

Представление данных: словарь (10)

```
// Проверка на пустоту
bool dict_is_empty(tdDict* d) {
    return d->size == 0;
}
```

Представление данных: словарь (11)

```
// Удаление пары по ключу
void dict_remove(tdDict* d, void* key) {
 tdPair* current = d->head; tdPair* prev = NULL;
 while (current) {
  if (current->key == key ||
   !strcmp((char*)current->key, (char*)key)) {
   if (prev) prev->next = current->next;
   else d->head = current->next;
   free(current); d->size--; return; }
  prev = current; current = current->next; }
}
```

Представление данных: словарь (12)

```
dict_put(d, (void*)"name", (void*)"Alice");
dict_put(d, (void*)"age", (void*)(intptr_t)30);
printf("Name: %s\n", (char*)dict_get(d,
(void*)"name"));
printf("Age: %d\n", (int)(intptr_t)dict_get(d,
(void*)"age"));
printf("Contains 'name': %s\n", dict_cont_key(d,
(void*)"name") ? "Yes" : "No");
dict_remove(d, (void*)"age");
printf("Contains 'age': %s\n", dict_cont_key(d,
(void*)"age") ? "Yes" : "No");
```

Представление данных: словарь (13)

```
Name: Alice
Age: 30
Contains 'name': Yes
Contains 'age': No
```

Представление данных: очередь (1)

```
// Структура очереди
typedef struct {
 tdNode* head; // Первый элемент
 tdNode* tail;  // Последний элемент
 int size;      // Размер очереди
} tdQueue;
```

Представление данных: очередь (2)

```
// Структура узел
typedef struct tdNode {
 void* data; // Данные
 struct tdNode* next; // Следующий узел
} tdNode;
```

Представление данных: очередь (3)

```
// Создание пустой очереди
tdQueue* queue_create() {
 tdQueue* q =
    (tdQueue*)malloc(sizeof(tdQueue));
 q->head = NULL; q->tail = NULL;
 q->size = 0; return q;
}
```

Представление данных: очередь (4)

```
// Освобождение памяти
void queue_free(tdQueue* q) {
 tdNode* curr = q->head;
 while (curr) {
  tdNode* temp = curr;
  curr = curr->next; free(temp); }
 free(q);
}
```

Представление данных: очередь (5)

```
int main()
{
 // Создание очереди
 tdQueue* q = queue_create();
 // Освобождение памяти
 queue_free(q);
 return EXIT_SUCCESS;
}
```

Представление данных: очередь (6)

```
// Добавление элемента в конец очереди
void queue_enqueue(tdQueue* q, void* d)
{
 tdNode* new_node =
    (tdNode*)malloc(sizeof(tdNode));
 new_node->data = data;
 new_node->next = NULL;
```

Представление данных: очередь (7)

```
if (q->tail == NULL) {
 q->head = new_node;
 q->tail = new_node; }
else {
 q->tail->next = new_node;
 q->head = new_node;}
q->size++;}
```

Представление данных: очередь (8)

```
// Удаление элемента из начала очереди
void* queue_dequeue(tdQueue* q) {
 tdNode* temp = q->head;
 void* data = temp->data;
 q->head = q->front->next;
 free(temp); q->size--;
 return data;
}
```

Представление данных: очередь (9)

```
// Проверка на пустоту
bool queue_is_empty(tdQueue* q) {
    return q->head == NULL;
}
// Получение размера очереди
int queue_size(tdQueue* q) {
    return q->size;
}
```

Представление данных: очередь (10)

```
queue_enqueue(q, (void*)(intptr_t)10);
queue_enqueue(q, (void*)(intptr_t)20);
queue_enqueue(q, (void*)(intptr_t)30);
printf("Queue size: %d\n", queue_size(q));
printf("Dequeued: %p\n", queue_dequeue(q));
printf("Dequeued: %p\n", queue_dequeue(q));
if (queue_is_empty(q)) printf("Queue is empty\n");
else printf("Queue is not empty\n");
printf("Queue size: %d\n", queue_size(q));
```

Представление данных: очередь (11)

```
Queue size: 3
Dequeued: 0000000000000000A
Dequeued: 0000000000000014
Queue is not empty
Queue size: 1
```

Представление данных: вектор (1)

```
// Структура вектора
typedef struct {
 void** data;  // Массив указателей
 int size;     // Текущий размер вектора
 int cap; // Емкость вектора
} Vector;
```

Представление данных: вектор (2)

```
// Создание вектора с начальной емкостью
tdVector* vector_create(int initCa) {
 tdVector* v =
    (tdVector*)malloc(sizeof(tdVector));
 v->data = (void**)malloc(initCa *
                          sizeof(void*));
 v->size = 0;  v->cap = initCa;
 return v; }
```

Представление данных: вектор (3)

```
// Освобождение памяти
void vector_free(tdVector* v) {
    if (v->data) {
        free(v->data);
    }
    free(v);
}
```

Представление данных: вектор (4)

```
int main()
{
    // Создание вектора 2х элементов
    Vector* v = vector_create(2)
    // Освобождение памяти
    vector_free(v);
    return EXIT_SUCCESS;
}
```

Представление данных: вектор (5)

```
// Добавление элемента в конец вектора
void vector_push(tdVector* v, void* el){
if (v->size >= v->cap) {
 int newCa= v->cap * 2;
 void** new_data = (void**)realloc(
v->data, newCa * sizeof(void*));
 v->data = new_data; v->cap = newCa; }
v->data[v->size] = el; v->size++; }
```

Представление данных: вектор (6)

```
// Удаление и возврат последнего элемента
void* vector_pop(tdVector* v) {
 if (!v || v->size == 0) return NULL;
 void* el = v->data[v->size - 1];
 v->size--;
 return el;
}
```

Представление данных: вектор (7)

```
// Получение элемента по индексу
void* vector_get(tdVector* v, int index)
{
 if (!v || index < 0 ||
          index >= v->size) return NULL;
 return v->data[index];
}
```

Представление данных: вектор (8)

```
vector_push(v, (void*)(intptr_t)10);
vector_push(v, (void*)(intptr_t)20);
vector_push(v, (void*)(intptr_t)30);
for (size_t i = 0; i < 2; i++)
 printf("Index %d: %p\n", i, vector_get(v, i));
printf("Popped element: %p\n", vector_pop(v));
for (size_t i = 0; i < 2; i++)
 printf(“Index %d: %p\n", i, vector_get(v, i));
```

Представление данных: вектор (9)

```
Index 0: 000000000000000A
Index 1: 0000000000000014
Index 2: 000000000000001E
Popped element: 000000000000001E
Index 0: 000000000000000A
Index 1: 0000000000000014
Index 2: 0000000000000000
```

Представление данных: список (1)

```
// Структура списка
typedef struct {
 tdsNode* head;     // Первый узел
    tdsNode* tail; // Последний узел
    int size;       // Размер списка
} tdList;
```

Представление данных: список (2)

```
// Структура узла списка
typedef struct tdsNode {
 void* data; // Данные
 struct tdsNode* next; // Следующий узел
} tdsNode;
```

Представление данных: список (3)

```
// Создание пустого списка
tdList* list_create() {
 tdList* l =
(tdList*)malloc(sizeof(tdList));
 l->head = NULL; l->tail = NULL;
 l->size = 0; return l;
}
```

Представление данных: список (4)

```
// Освобождение памяти
void list_free(tdList* l) {
 tdsNode* current = l->head;
 while (current) {
  tdsNode* temp = current;
  current = current->next;
  free(temp); }
 free(l); }
```

Представление данных: список (5)

```
int main(){
    // Создание списка
    tdList* l = list_create();
    // Освобождение памяти
    list_free(l);
    return EXIT_SUCCESS;
}
```

Представление данных: список (6)

```
// Добавление элемента в начало списка
void list_pushf(tdList* l, void* d) {
 tdsNode* new_node =
            (tdsNode*)malloc(sizeof(tdsNode));
 new_node->data = d;
 new_node->next = l->head;
 l->head = new_node;
 if (l->size == 0) l->tail = new_node;
 l->size++;}
```

Представление данных: список (7)

```
// Добавление элемента в конец списка
void list_pushb(tdList* l, void* d) {
 tdsNode* new_node =
(tdsNode*)malloc(sizeof(tdsNode));
 new_node->data = d; new_node->next = NULL;
 if (l->size == 0) { l->head = new_node;
 l->tail = new_node; } else { l->tail->next =
new_node; l->tail = new_node;
 } l->size++; }
```

Представление данных: список (8)

```
// Удаление первого элемента из списка
void* list_popf(tdList* l) {
 tdsNode* temp = l->head;
 void* data = temp->data;
 l->head = l->head->next;
 if (l->head == NULL)l->tail = NULL;
 free(temp); l->size--;
 return data; }
```

Представление данных: список (9)

```
// Удаление последнего элемента из списка
void* list_popb(tdList* l) {
 tdsNode* curr = l->head; tdsNode* prev = NULL;
 while (curr && curr >next) {
  prev = curr; curr = curr->next; }
 void* data = curr->data;
  if (prev){prev->next = NULL;
 l->tail = prev; } else {
 l->head = NULL; l->tail = NULL; }
 free(curr); l->size--; return data; }
```

Представление данных: список (10)

```
// Проверка на пустоту
bool list_is_empty(tdList* l) {
    return l->size == 0;
}
```

Представление данных: список (11)

```
list_push_front(l, (void*)(intptr_t)10);
list_push_back(l, (void*)(intptr_t)20);
list_push_back(l, (void*)(intptr_t)30);
printf("Popped from front: %p\n",
                          list_pop_front(l));
printf("Popped from back: %p\n",
                          list_pop_back(l));
if (list_is_empty(l)) printf("List is empty\n");
else printf("List is not empty\n");
```

Представление данных: список (12)

```
Popped from front: 000000000000000A
Popped from back: 000000000000001E
List is not empty
```

Спасибо за внимание

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)