Last updated: 2019-02-10
vadimov@i.ua

Практическое занятие 1 (семестр2).

В приведенном в Unit 1 "Growing Arrays" примере определяется растущий массив элементов типа Nameval; новые элементы добавляются в конец массива, который расширяется по мере необходимости.

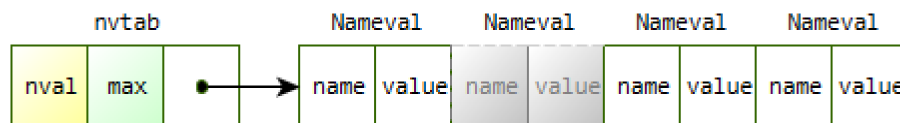
Задача 1.1. Переменная struct NVtab {...} nvtab является глобальной. Модифицируйте функции addname и delname (см. Unit 1 "Growing Arrays") так, чтобы в теле этих функций устранить использование глобальной переменной nvtab.

Задача 1.2. В коде Unit 1 "Growing Arrays" функция delname не вызывает функцию realloc, чтобы возвратить память, освободившуюся при удалении. Стоит ли это делать? Как бы вы определили, делать это или нет?

Задача 1.3. Поскольку адрес массива может изменяться при перераспределении памяти, в основной части программы (в теле функции main()) следует обращаться к элементам по индексам, а не через указатели. Почему? Покажите пример обращения к элементам через указатели.

Задача 1.4. Обратите внимание, что в коде Unit 1 "Growing Arrays" не употребляется такая конструкция:
nvtab.nameval = (Nameval *) realloc(nvtab.nameval, (NVGROW*nvtab.max) * sizeof(Nameval));
В этом случае, если бы при перераспределении памяти произошла ошибка, все накопленные в исходном массиве данные были бы потеряны. Объясните почему? Покажите действующий пример.

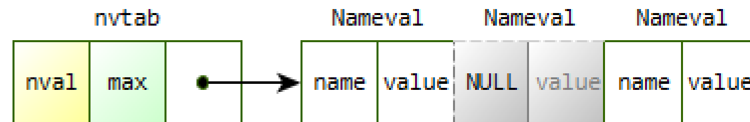
Задача 1.5. Удаление имени из массива требует решить что делать с освободившейся ячейкой. Если порядок элементов не имеет значения, то легче всего перебросить последний элемент в образовавшуюся свободную позицию. Если же порядок необходимо соблюсти, то придется сдвинуть на одну ячейку назад все элементы после свободной позиции. Именно так поступает функция delname (см. Unit 1 "Growing Arrays").



Покажите на действующем примере переброску последнего элемента в образовавшуюся свободную позицию.

Задача 1.6. Внесите необходимые изменения в код функции addname и delname (см. Unit 1 "Growing Arrays") так, чтобы удалять имена, помечая их как неиспользуемые. Насколько

остальная часть программы изолирована от этих изменений?
Подсказка: Подход, альтернативный перемещению элементов (memmove) в массиве, - это пометить удаленные элементы как неиспользуемые. Затем, чтобы добавить новый элемент, вначале необходимо найти свободную ячейку, и только в том случае, если таковых нет, расширить массив до нового размера. В примере (см. Unit 1 "Growing Arrays"), чтобы пометить элемент как неиспользуемый, в его поле name можно записать значение NULL.



Задача 1.7. Напишите функцию для вывода всех элементов списка (см. Unit 1 "A singly-linked list"), которая перебирает его и последовательно выводит каждый элемент.
Подсказка: Чтобы вычислить длину списка, можно написать функцию с простым перебором и инкрементированием счетчика.

Задача 1.8. Напишите функцию coolfun, выполняющую перебор списка (см. Unit 1 "A singly-linked list") и вызывающую другую заданную функцию для каждого его элемента. Функцию coolfun сделать гибкой, включив в ее параметры аргумент для передачи в ту, другую функцию. Таким образом, coolfun будет принимать три аргумента: сам список; указатель на функцию, вызываемую для каждого элемента списка; аргумент для передачи в эту функцию. Прототип: `void coolfun(Nameval *listp, void (*fn)(Nameval*, void*), void *arg).`
Пример. Для подсчета элементов определяется функция с аргументом в виде указателя на целочисленный счетчик, который нужно инкрементировать:

```
/* inccounter: инкрементирует счетчик *arg */
void inccounter(Nameval *p, void *arg) {
    int *ip;
    /* p здесь не используется! */
    ip = (int *)arg;
    (*ip)++;
}
```

Эта функция вызывается следующим образом:

```
int n = 0;
coolfun(nvlist, inccounter, &n);
printf("%d elements in nvlist\n", n);
```

Задача 1.9. Реализуйте некоторые другие возможные операции со списком (см. Unit 1 "A singly-linked list"), а именно: копирование, слияние, разбиение, вставку перед конкретным элементом или после него. Как две различные операции вставки отличаются друг от друга по сложности? В какой степени можно воспользоваться приведенными в лекции

функциями, а что придется написать самостоятельно?

Задача 1.10. Напишите рекурсивную и итерационную версии функции reverse_list, которая бы изменяла порядок следования элементов в списке на противоположный. Не создавайте новых элементов списка — используйте только существующие.

Задача 1.11. Как и функция freeall (см. Unit 1 "A singly-linked list"), функция delitem не освобождает поля name. Модифицируйте обе эти функции с учетом того, что память, выделенную под поля name, нужно освобождать.

Задача 1.12. Напишите нетипизированное определение типа Type_List на языке C. Самый простой способ - это включить в состав каждого элемента списка указатель на данные типа void*. Только для продвинутых: проделайте то же самое на языке C++ в виде шаблона, а также на Java, определив класс для списка, содержащего элементы типа Object. Каковы сильные и слабые стороны разных языков в реализации этой задачи?

Задача 1.13. Придумайте и реализуйте набор тестов для проверки правильности написанных вами функций работы со списками.

Задача 1.14. Дерево, в котором любой путь от корня к листу имеет примерно одинаковую длину, называется сбалансированным. (Преимущество сбалансированного дерева заключается в том, что поиск по нему имеет характер 0(log n), поскольку, как и в двоичном поиске, на каждом шаге отбрасывается половина оставшихся данных.) Сгенерировать поток данных для заполнения дерева (см. Unit 1 "Binary search tree"). По мере поступления данных для каждого элемента данных формировать узел и добавлять его в дерево.
Вариант "1": элементы поступают в случайном порядке (поступающие данные достаточно стохастичны). Написать функцию проверки сбалансированности заполненного (построенного) дерева, чтобы подтвердить или опровергнуть следующее утверждение: если элементы поступают в случайном порядке, то дерево будет более-менее сбалансированным.
Вариант "2": элементы прибывают в отсортированном виде. Написать функцию проверки сбалансированности заполненного (построенного) дерева, чтобы подтвердить или опровергнуть следующее утверждение: если элементы прибывают в отсортированном виде, то спуск всегда будет выполняться до самого низа одной из ветвей дерева, фактически представляя собой список по указателю right. Этот случай характеризуется проблемами быстродействия, присущими спискам.

Задача 1.15. Сравните быстродействие функций lookup и nvlookup (см. Unit 1 "Binary search tree"). Какова разница между рекурсивной и итерационной формами?

Задача 1.16. Напишите функцию сортировки с симметричным обходом (см. Unit 1 "Tree-traverser") . Какой порядок по быстродействию имеет данная операция? При каких условиях она может работать плохо? Каковы ее характеристики по сравнению с алгоритмом быстрой сортировки и с библиотечными функциями?

Задача 1.17. Придумайте и реализуйте набор тестов для проверки правильности функций работы с деревьями, рассмотренных в Unit 1 "Binary search tree" и "Tree-traverser".

Задача 1.18. Каким должен быть размер массива symtab (см. Unit 1 "Hash Table")? Общая идея состоит в том, что массив symtab должен быть достаточно велик, чтобы цепочка каждого хэш-кода содержала поменьше элементов, и операция поиска имела характер 0(1). Свой ответ подкрепите примером кода.

Задача 1.19. Хэш-функция, хорошо работающая с данными одного вида (например, короткими именами переменных), может оказаться неудачной в работе с другими (такими как URL-адреса), поэтому хэш-функцию для своей программы следует тестировать на типичных наборах входных данных. Хорошо ли она кодирует короткие строки? А длинные? А строки одинаковой длины с небольшими отличиями? Сгенерируйте поток данных для заполнения хэша и проверьте качество хэширования (длины цепочек) при MULTIPLIER от 31 до 37.

Задача 1.20. Хэш-функция (см. Unit 1 "Hash Table") имеет довольно общий характер и удобна при работе со строками. Однако с некоторыми исходными данными она может справляться недостаточно эффективно. Сконструируйте набор данных, который бы заставил эту функцию работать плохо. Насколько трудно построить такой набор для различных значений NHASH?

Задача 1.21. Напишите функцию для обращения к последовательным элементам хэш-таблицы в несортированном порядке.

Задача 1.22. Модифицируйте функцию lookup (см. Unit 1 "Hash Table") так, чтобы при превышении средней длиной списка некоторого порога x массив расширялся бы автоматически с коэффициентом пропорциональности y и чтобы хэш-таблица подвергалась перестройке.

Задача 1.23. Для продвинутых. Разработайте свою хэш-функцию для хранения координат точек в 2-мерном пространстве. Насколько легко адаптировать вашу функцию к изменениям типа координат (например, от целочисленных к вещественным), системы координат (от декартовой к полярной) или размерности (от двух к более высокой)? Например, можно хэш-кодировать 2-мерные координаты точек, тем самым организуя хранилище данных в виде линейной таблицы порядка 0(количество точек) вместо 2-мерного массива порядка 0(размерX * размерY).

Важно: ответы на поставленные в каждой задаче вопросы, оформить в виде комментария в конце реализующего решение задачи кода.