**ПРАКТИЧНЕ ЗАВДАННЯ № 3**

**Функції мови**

***Частина перша***

**Завдання № 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.

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval {

char \*name;

int value;

} Nameval;

struct Array { //NVtab

int nval;

int max;

Nameval \*items;

};

enum { NVINIT = 1, NVGROW = 2 };

int add\_item(struct Array \*array, Nameval newname){

Nameval \*temp;

if (array->items == NULL) {

array->items = (Nameval \*)malloc(NVINIT \* sizeof(Nameval));

if(array->items == NULL)

return -1;

array->max = NVINIT;

array->nval = 0;

}

else if (array->nval >= array->max) {

temp = (Nameval \*) realloc(array->items,

(NVGROW \* array->max) \* sizeof(Nameval));

if (temp == NULL)

return -1;

array->max \*= NVGROW;

array->items = temp;

}

array->items[array->nval] = newname;

return ++array->nval;

}

int del\_item(struct Array \*array, char \*name) {

for (int i = 0; i<array->nval; ++i){

if (strcmp(array->items[i].name, name) == 0){

memmove(array->items + i, array->items + i + 1, (array->nval - (i + 1)) \* sizeof(Nameval));

--array->nval;

return 1;

}

}

return 0;

}

int main() {

struct Array array = { 0, 0, NULL }; //Initializing fields of a structure

add\_item(&array, (struct Nameval) {.name = "Andy", .value = 12});

add\_item(&array, (struct Nameval){.name = "Billy", .value = 16});

add\_item(&array, (struct Nameval){.name = "Tom", .value= 25});

add\_item(&array, (struct Nameval){.name = "Alice", .value= 25});

for (int i = 0; i < array.nval; ++i){

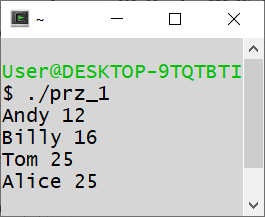
printf("%s %d\n", array.items[i].name, array.items[i].value);

}

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 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?

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval {

char \*name;

int value;

} Nameval;

struct Array { //NVtab

int nval;

int max;

Nameval \*items;

};

enum { NVINIT = 1, NVGROW = 2 };

int add\_timem(struct Array \*array, Nameval newname){

Nameval \*temp;

if (array->items == NULL) {

array->items = (Nameval \*)malloc(NVINIT \* sizeof(Nameval));

if(array->items == NULL)

return -1;

array->max = NVINIT;

array->nval = 0;

}

else if (array->nval >= array->max) {

temp = (Nameval \*) realloc(array->items,

(NVGROW \* array->max) \* sizeof(Nameval));

if (temp == NULL)

return -1;

array->max \*= NVGROW;

array->items = temp;

}

array->items[array->nval] = newname;

return ++array->nval;

}

int delname(struct Array \*array, char \*name) {

for (int i = 0; i<array->nval; ++i){

if (strcmp(array->items[i].name, name) == 0){

memmove(array->items + i, array->items + i + 1, (array->nval - (i + 1)) \* sizeof(Nameval));

--array->nval;

return 1;

}

}

return 0;

}

int main() {

struct Array array = { 0, 0, NULL }; //Initializing fields of a structure

add\_timem(&array, (struct Nameval) {.name = "Andy", .value = 12});

add\_timem(&array, (struct Nameval){.name = "Billy", .value = 16});

add\_timem(&array, (struct Nameval){.name = "Tom", .value= 25});

add\_timem(&array, (struct Nameval){.name = "Alice", .value= 25});

for (int i = 0; i < array.nval; ++i){

printf("%s %d\n", array.items[i].name, array.items[i].value);

}

return 0;

}

//Функція delname не викликає функцію realloc, адже:

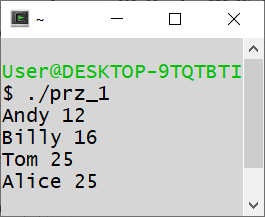
//функція realloc()повертає вказівник на перший байт знову

//виділеного блоку памяті, а оскільки в програмі не передбачене

//подальше використання цього блоку пам'яті, то її використання

// не є необхідністю.

**Результати роботи програми (скриншоти):**



**Завдання № 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.

**Код програми:**

//функція realloc() - це функція, яка використовується тоді, коли

//потрібно змінити розмір виділеної пам'яті (як на більший, так і наменьший).

//Ця функція може використовувати як і раніше виділену ділянку пам'яті,

//так і нову

#include <stdio.h>

#define SIZE 40

struct tag {

char name[SIZE];

int episodes;

char duration[SIZE];

};

int main(){

struct tag film =

{"The Queen's Gambit",7,"3 hours, 33 minutes"};

struct tag \*ptr;

ptr = &film;

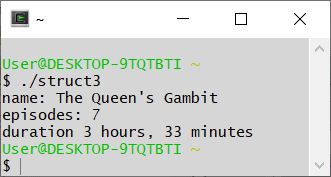
printf("name: %s\nepisodes: %d\nduration %s",

ptr->name, ptr->episodes,ptr->duration);

return 0;

}

**Результати роботи програми (скриншоти):**



**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval {

char \*name;

int value;

} Nameval;

struct Array {

int nval;

int max;

Nameval \*items;

};

enum { NVINIT = 1, NVGROW = 2 };

int add\_item(struct Array \*array, Nameval newname){

Nameval \*temp;

if (array->items == NULL) {

array->items = (Nameval \*)malloc(NVINIT \* sizeof(Nameval));

if(array->items == NULL)

return -1;

array->max = NVINIT;

array->nval = 0;

}

else if (array->nval >= array->max) {

temp = (Nameval \*) realloc(array->items,

(NVGROW \* array->max) \* sizeof(Nameval));

if (temp == NULL)

return -1;

array->max \*= NVGROW;

array->items = temp;

}

array->items[array->nval] = newname;

return ++array->nval;

}

int del\_item(struct Array \*array, char \*name) {

for (int i = 0; i<array->nval; ++i){

if (strcmp(array->items[i].name, name) == 0){

memmove(array->items + i, array->items + i + 1, (array->nval - (i + 1)) \* sizeof(Nameval));

--array->nval;

return 1;

}

}

return 0;

}

int main() {

struct Array array = { 0, 0, NULL }; //Initializing fields of a structure

add\_item(&array, (struct Nameval) {.name = "Andy", .value = 12});

add\_item(&array, (struct Nameval){.name = "Billy", .value = 16});

add\_item(&array, (struct Nameval){.name = "Tom", .value= 25});

add\_item(&array, (struct Nameval){.name = "Alice", .value= 25});

for (int i = 0; i < array.nval; ++i){

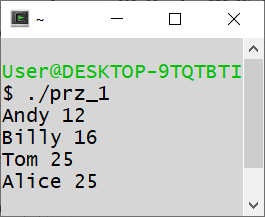
printf("%s %d\n", array.items[i].name, array.items[i].value);

}

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 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.

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval {

char \*name;

int value;

} Nameval;

struct Array {

int lenght;

int capacity;

Nameval \*items;

};

enum {NVINIT = 1, NVGROW = 2};

int add\_item(struct Array \*Array, Nameval newname) {

// Nameval \*nvp;

if (Array->items == NULL) {

Array->items = (Nameval \*) malloc(NVINIT \* sizeof(Nameval));

if (Array->items == NULL)

return -1;

Array->capacity = NVINIT;

Array->lenght = 0;

} else if (Array->lenght >= Array->capacity) {

// vp = (items \*) realloc(Array->items, (NVGROW\*Array->capacity)\*sizeof(Nameval));

Array->items = (Nameval \*) realloc(Array->items, (NVGROW\*Array->capacity)\*sizeof(Nameval));

if (Array->items == NULL) return -1;

Array->capacity \*= NVGROW;

// Array->items = nvp;

}

Array->items[Array->lenght] = newname;

return Array->lenght++;

}

int del\_item(struct Array \*Array, char \*name) {

for (int i = 0; i < Array->lenght; ++i) {

if (strcmp(Array->items[i].name, name) == 0){

memmove(Array->items + i, Array->items + i + 1, (Array->lenght - ( i + 1))\*sizeof(Nameval));

Array->lenght--;

return 1;

}

}

return 0;

}

int main () {

struct Array Array = {0, 0, NULL}; //Initializing fields of a structure

add\_item (&Array, (struct Nameval) {.name = "Andy", .value = 12});

add\_item(&Array, (struct Nameval) {.name = "Billy", .value = 18});

for (int i = 0; i < Array.lenght; ++i) {

printf("%s %d\n", Array.items[i].name, Array.items[i].value);

}

return 0;

}

//Функція void \*realloc(void \*ptr, size\_t size);

//повертає вказівник на перший байт знову виділеного блоку пам'яті

//і при цьому копіює вміст попереднього блоку пам'яті в новий

//тому і не відбувається втрата пам'яті.

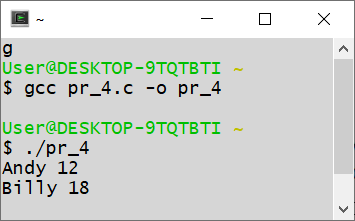
//В конструкції

//nvtab.nameval = (Nameval \*) realloc(nvtab.nameval, (NVGROW\*nvtab.max) \* sizeof(Nameval));

//копіювання буде відбуватись у той самий блок пам'яті

//через що можлива втрата даних.

**Результати роботи програми (скриншоти):**



**Завдання № 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.

**Код програми**

**a)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval {

char \*name;

int value;

} Nameval;

struct Array {

int lenght;

int capacity;

Nameval \*items;

} array;

enum { NVINIT = 1, NVGROW = 2 };

int addname(Nameval newname) {

Nameval \*temp;

if (array.items == NULL) {

array.items = (Nameval \*)malloc(NVINIT \* sizeof(Nameval));

if (array.items == NULL)

return -1;

array.capacity = NVINIT;

array.lenght = 0;

}

else if (array.lenght == array.capacity) {

temp = (Nameval \*)realloc(array.items, (16 \* array.capacity) \* sizeof(Nameval));

if (temp == NULL)

return -1;

array.capacity \*= NVGROW;

array.items = temp;

}

array.items[array.lenght] = newname;

return array.lenght++;

}

int delname(char \*name) {

for (int i = 0; i < array.lenght; i++){

if (strcmp(array.items[i].name, name) == 0){

memmove(array.items + i, array.items + i + 1, (array.lenght - (i + 1)) \* sizeof(Nameval));

//for (int j = i; j < array.lenght; j++) ///заміна функції memmove(),

// array.items[j] = array.items[j + 1];

array.lenght--;

return 1;

}

}

return 0;

}

int main() {

// struct Array array = {0, 0, NULL};

addname((struct Nameval){.name = "Andy", .value = 12});

addname((struct Nameval){.name = "Billy", .value = 17});

addname((struct Nameval){.name = "Mike", .value = 17});

addname((struct Nameval){.name = "Endy", .value = 17});

addname((struct Nameval){.name = "John", .value = 17});

for(int i = 0; i < array.lenght; ++i)

printf("%s %d\n", array.items[i].name, array.items[i].value);

delname("Endy");

for(int i = 0; i < array.lenght; ++i)

printf("%s %d\n", array.items[i].name, array.items[i].value);

return 0;

}

**b)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct {

char \*name;

int value;

} Nameval;

typedef struct {

Nameval \*items;

int length;

int capacity;

} Array;

Array\* create() {

Array \*result = malloc(sizeof(Array));

result->items = malloc(16 \* sizeof(Nameval));

result->length = 0;

result->capacity = 16;

return result;

}

void destroy(Array \*array) {

if (array != NULL && array->items != NULL) {

for (int i = 0; i < array->length; ++i) {

free(array->items[i].name);

}

free(array->items);

}

free(array);

}

void expand(Array \*array) {

array->items = realloc(array->items, array->capacity \*= 2);

}

void resize(Array \*array, int size) {

while (array->capacity < size) {

expand(array);

}

array->length = size;

}

void add\_item(Array \*array, Nameval item) {

resize(array, array->length + 1);

array->items[array->length - 1] = item;

}

void remove\_item(Array \*array, int position) {

if (position < 0 || position >= array->length) {

return;

}

array->items[position] = array->items[array->length - 1];

resize(array, array->length - 1);

}

void remove\_items\_by\_name(Array \*array, const char \*name) {

for (int i = array->length - 1; i >= 0; --i) {

if (strcmp(array->items[i].name, name) == 0) {

remove\_item(array, i);

}

}

}

char\* make\_string(const char \*source) {

return strcpy(malloc(sizeof(char) \* (strlen(source) + 1)), source);

}

Array\* init() {

Array \*result = create();

int size;

scanf("%d", &size);

for (int i = 0; i < size; ++i) {

static char name[1024];

int value;

scanf("%s %d", name, &value);

add\_item(result, (Nameval){.name = make\_string(name), .value = value});

}

return result;

}

void print(Array \*array) {

if (array->items == NULL) {

return;

}

for (int i = 0; i < array->length; ++i) {

printf("%s %d\n", array->items[i].name, array->items[i].value);

}

putchar('\n');

}

int main (void) {

Array \*array = init();

puts("\n\n");

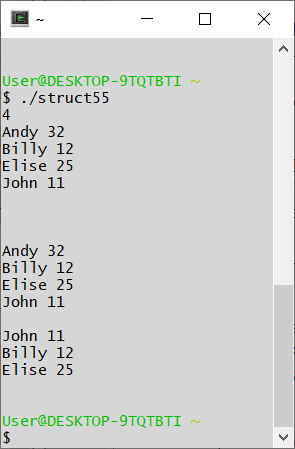
print(array);

remove\_items\_by\_name(array, "Andy");

print(array);

}

**Результати роботи програми (скриншоти):**



**Завдання № 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.



**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval Nameval;

struct Nameval {

char \*name;

int value;

};

struct NVtab {

int nval; int max;

Nameval \*nameval;

} nvtab;

enum {NVINIT = 1, NVGROW = 2};

int addname(Nameval newname){

Nameval \*nvp;

if (nvtab.nameval == NULL){

nvtab.nameval = (Nameval \*)malloc(NVINIT \* sizeof(Nameval));

if (nvtab.nameval == NULL)

return -1;

nvtab.max = NVINIT;

nvtab.nval = 0;

}

else{

for (int i = 0; i < nvtab.nval; ++i) {

if (nvtab.nameval[i].name == NULL) {

nvtab.nameval[i] = newname;

return ++nvtab.nval;

}

}

}

if (nvtab.nval == nvtab.max) {

nvp = (Nameval \*)realloc(nvtab.nameval, (NVGROW \* nvtab.max) \* sizeof(Nameval));

if (nvp == NULL)

return -1;

nvtab.max \*= NVGROW;

nvtab.nameval = nvp;

}

nvtab.nameval[nvtab.nval] = newname;

++return nvtab.nval;

}

int delname(char \*name) {

for (int i = 0; i < nvtab.nval; ++i) {

if (strcmp(nvtab.nameval[i].name, name) == 0) {

nvtab.nameval[i].name = NULL;

nvtab.nval--;

return 1;

}

}

return 0;

}

int main(){

addname((struct Nameval){.name = "Andy", .value = 12});

addname((struct Nameval){.name = "Billy", .value = 17});

addname((struct Nameval){.name = "Mike", .value = 17});

addname((struct Nameval){.name = "Endy", .value = 17});

addname((struct Nameval){.name = "John", .value = 17});

delname("Endy");

addname((struct Nameval){.name = "Alice", .value = 17});

for(int i = 0; i < nvtab.nval; ++i){

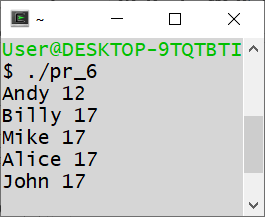
printf("%s %d\n", nvtab.nameval[i].name, nvtab.nameval[i].value);

}

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 7:**

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

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval Nameval;

struct Nameval {

char \*name;

int value;

Nameval \*next;

};

Nameval \*newitem(char \*name, int value){

Nameval \*newp;

newp = (Nameval \*)malloc(sizeof(Nameval));

newp->name = name;

newp->value = value;

newp->next = NULL;

return newp;

}

Nameval \*addfront(Nameval \*listp, Nameval \*newp){

newp->next = listp;

return newp;

}

Nameval \*addend(Nameval \*listp, Nameval \*newp){

Nameval \*p;

if (listp == NULL)

return newp;

for (p = listp; p->next != NULL; p = p->next);

p->next = newp;

return listp;

}

void inccounter(Nameval \*p, void \*arg) {

int \*ip;

ip = (int \*)arg;

(\*ip)++;

}

void show\_list(Nameval \*listp) {

Nameval \*ptr = listp;

for(int i = 0; ptr != NULL; ++i){

printf("%s %d\n", ptr->name, ptr->value);

ptr = ptr->next;

}

}

int main(){

Nameval \*nvlist = NULL;

int n = 0;

nvlist = addfront(nvlist, newitem("data1", 10));

Nameval \*item = newitem("data2", 45);

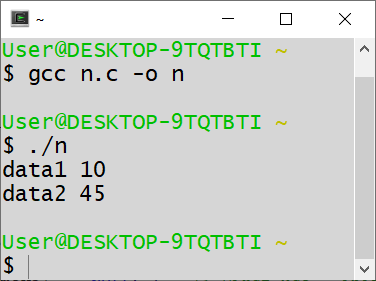
nvlist = addend(nvlist, item);

show\_list(nvlist);

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 8:**

Напишите функцию coolfun, выполняющую перебор списка (см. Unit 1 "A singlylinked 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);

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval Nameval;

struct Nameval {

char \*name;

int value;

Nameval \*next;

};

Nameval \*create\_item(char \*name, int value){

Nameval \*newp;

newp = (Nameval \*)malloc(sizeof(Nameval));

newp->name = name;

newp->value = value;

newp->next = NULL;

return newp;

}

Nameval \*push\_front(Nameval \*listp, Nameval \*newp){

newp->next = listp;

return newp;

}

Nameval \*push\_end(Nameval \*listp, Nameval \*newp){

Nameval \*p;

if (listp == NULL)

return newp;

for (p = listp; p->next != NULL; p = p->next);

p->next = newp;

return listp;

}

void inccounter(Nameval \*p, void \*arg) {

int \*ip;

ip = (int \*)arg;

(\*ip)++;

}

void show\_list(Nameval \*listp) {

Nameval \*p = listp;

while (p != NULL) {

printf("%s %d\n", p->name, p->value);

p = p->next;

}

}

void coolfun(Nameval \*listp, void (\*fn)(Nameval\*, void\*), void \*arg){

for ( ;listp != NULL; listp = listp->next)

(\*fn)(listp, arg);

}

int main(){

Nameval \*list = NULL;

int n = 0;

list = push\_front(list, create\_item("data1", 10));

Nameval \*item = create\_item("data2", 45);

list = push\_end(list, item);

show\_list(list);

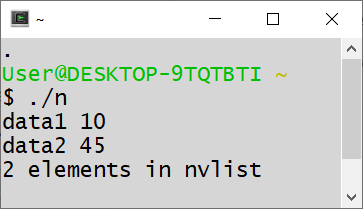
coolfun(list, inccounter, &n);

printf("%d elements in list\n", n);

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 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?

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct List List;

struct List {

char \*name;

int value;

List \*next;

};

List \*lookup(List \*listp, char \*name){

for ( ; listp != NULL; listp = listp->next)

if (strcmp(name, listp->name) == 0)

return listp;

return NULL;

}

List \*create\_item(char \*name, int value){

List \*newp;

newp = (List \*)malloc(sizeof(List));

newp->name = name;

newp->value = value;

newp->next = NULL;

return newp;

}

List \*push\_front(List \*listp, List \*newp){

newp->next = listp;

return newp;

}

List \*push\_end(List \*listp, List \*newp){

List \*p;

if (listp == NULL)

return newp;

for (p = listp; p->next != NULL; p = p->next);

p->next = newp;

return listp;

}

void free\_list(List \*listp){

List \*next;

for ( ; listp != NULL; listp = next) {

next = listp->next;

/\* assumes name is freed elsewhere \*/

free(listp);

}

}

List \*del\_item(List \*listp, char \*name){

List \*p, \*prev;

prev = NULL;

for (p = listp; p != NULL; p = p->next) {

if (strcmp(name, p->name) == 0) {

if (prev == NULL)

listp = p->next;

else

prev->next = p->next;

free(p);

return listp;

}

prev = p;

}

return NULL;

}

void print(const List \*listp){

const List \*p;

for (p = listp; p != NULL; p = p->next) {

printf("%s: %d ", p->name, p->value);

}

}

List \*create\_item1(char \*name, int value) {

List \*newp;

newp = (List \*)malloc(sizeof(List));

newp->name = name;

newp->value = value;

return newp;

}

/\* int lists\_equal(const List \*lista, const List \*listb) {

for( ; lista != NULL && listb != NULL; lista->next, listb->next) {

if (lista->value != listb->value || strcmp(lista->name, listb->name) != 0)

return 0;

}

return lista == NULL && listb == NULL;

}\*/

List \*copy(const List \*listp) {

if(listp == NULL)

return 0;

List \*newp = create\_item1(listp->name, listp->value);

newp->next = copy(listp->next);

return newp;

}

List \*merge(List \*lista, List \*listb){

List \*p;

if (lista == NULL)

return listb;

for (p = lista; p->next != NULL; p = p->next)

continue;

p->next = listb;

return lista;

}

List \*split(List \*listp, char \*name, List \*\*other){

List \*p, \*prev;

prev = NULL;

for (p = listp; p != NULL; p = p->next) {

if (strcmp(p->name, name) == 0) {

\*other = p;

if (prev == NULL)

listp = NULL;

else

prev->next = NULL;

return listp;

}

prev = p;

}

\*other = NULL;

return listp;

}

List \*insert\_before(List \*listp, List \*item, List \*newp){

List \*p, \*prev;

prev = NULL;

for (p = listp; p != NULL; p = p->next) {

if (p == item) {

p = push\_front(item, newp);

if (prev == NULL)

listp = p;

else

prev->next = p;

return listp;

}

prev = p;

}

return NULL;

}

List \*insert\_after(List \*listp, List \*item, List \*newp){

item->next = push\_front(item->next, newp);

return listp;

}

int main(){

List \*listp = NULL, \*listp1 = NULL, \*merged = NULL;

List \*other;

// printf("original\n");

listp = push\_front(listp, create\_item("Andy", 2));

listp = push\_front(listp, create\_item("Billy",8));

listp = push\_front(listp, create\_item("Alice", 5));

listp = push\_front(listp, create\_item("John", 4));

print(listp);printf("\n\n");

// printf("insert\n");

listp = insert\_before(listp, listp, create\_item("Nik", 3));

listp = insert\_before(listp, lookup(listp, "John"), create\_item("Ted", 7));

insert\_after(listp, lookup(listp, "Alice"), create\_item("Mike", 6));

del\_item(listp, "Andy");

print(listp); printf("\n\n");

// printf("copy\n");

listp1 = copy(listp);

print(listp1); printf("\n\n");

// printf("split\n");

listp = split(listp, "Alice", &other);

print(listp); printf("\n\n");

// printf("merge\n");

merged = merge(listp1, listp);

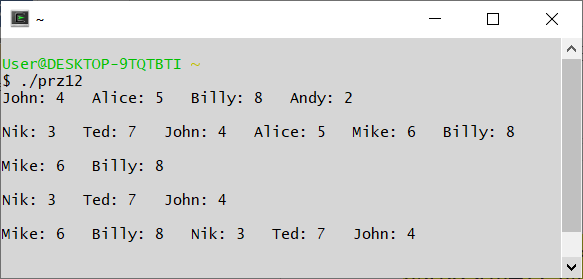
print(merged); printf("\n\n");

return 0;

}

//зазвичай вставка до якогось елемента має складність О(n), а встакавка після О(n+1), що означає, що вона є складнішою, тому можна і варто очистити пам'ять після її використання

**Результати роботи програми (скриншоти):**



**Завдання № 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.

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval Nameval;

struct Nameval {

char \*name;

int value;

Nameval \*next;

};

Nameval \*newitem(char \*name, int value){

Nameval \*newp;

newp = (Nameval \*)malloc(sizeof(Nameval));

newp->name = name;

newp->value = value;

newp->next = NULL;

return newp;

}

Nameval \*addfront(Nameval \*listp, Nameval \*newp){

newp->next = listp;

return newp;

}

Nameval \*addend(Nameval \*listp, Nameval \*newp){

Nameval \*p;

if (listp == NULL)

return newp;

for (p = listp; p->next != NULL; p = p->next);

p->next = newp;

return listp;

}

void show\_list(struct Nameval \*listp) {

struct Nameval \*temp = listp;

while(temp != NULL) {

printf("%s %d\n", temp->name, temp->value);

temp = temp->next;

}

}

void reverse(struct Nameval\*\* listp) {

struct Nameval\* prev = NULL;

struct Nameval\* current = \*listp;

struct Nameval\* next;

while (current != NULL)

{

next = current->next;

current->next = prev;

prev = current;

current = next;

}

\*listp = prev;

}

void recursiveReverse(struct Nameval \*\*listp){

struct Nameval\* first;

struct Nameval\* oters;

if (\*listp == NULL)

return;

first = \*listp;

oters = first->next;

if (oters == NULL)

return;

recursiveReverse(&oters);

first->next->next = first;

first->next = NULL;

\*listp = oters;

}

int main(){

Nameval \*nvlist = NULL;

int n = 0;

nvlist = addfront(nvlist, newitem("data1", 10));

nvlist = addend(nvlist, newitem("data2", 45));

nvlist = addend(nvlist, newitem("data3", 40));

show\_list(nvlist);

puts("\n\n");

reverse(&nvlist);

show\_list(nvlist);

puts("\n\n");

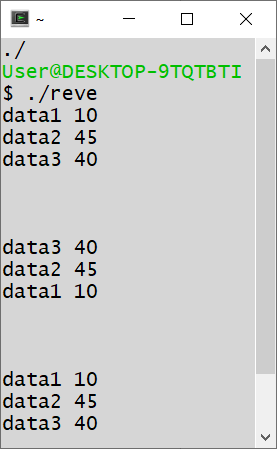
recursiveReverse(&nvlist);

show\_list(nvlist);

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 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.

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval Nameval;

struct Nameval {

char \*name;

int value;

Nameval \*next;

};

Nameval \*create\_item(char \*name, int value) {

Nameval \*new;

new = (Nameval\*)malloc(sizeof(Nameval));

new->name = malloc(strlen(name) +1);

strcpy(new->name, name);

new->value = value;

new->next = NULL;

return new;

}

Nameval \*puch\_front(Nameval \*listp, Nameval \*new){

new->next = listp;

return new;

}

Nameval \*push\_end(Nameval \*listp, Nameval \*new){

Nameval \*p;

if(listp == NULL)

return new;

for (p=listp; p->next !=NULL; p = p->next);

p->next = new;

return listp;

}

Nameval \*search(Nameval \*listp, char \*name) {

for ( ; listp != NULL; listp = listp->next)

if (strcmp(name, listp->name) == 0)

return listp;

return NULL;

}

Nameval \*delitem(Nameval \*listp, char \*name){

Nameval \*p, \*prev;

prev = NULL;

for (p = listp; p != NULL; p = p->next) {

if (strcmp(name, p->name) == 0) {

if (prev == NULL)

listp = p->next;

else

prev->next = p->next;

// free(p->name);

free(p);

return listp;

}

prev = p;

}

return NULL;

}

void print(Nameval \*listp) {

for( ; listp != NULL; listp = listp->next)

printf("%s %d\n", listp->name, listp->value);

printf("\n\n");

}

void freeall(Nameval \*listp){

Nameval \*next;

for( ; listp != NULL; listp = listp->next){

free(listp->name);

free(listp);

}

}

int main() {

Nameval \*nvlist = NULL;

nvlist = puch\_front(nvlist, create\_item("data1", 10));

nvlist = push\_end(nvlist, create\_item("data2", 45));

nvlist = push\_end(nvlist, create\_item("data3", 100));

nvlist = push\_end(nvlist, create\_item("data4", 50));

nvlist = push\_end(nvlist, create\_item("data5", 15));

print(nvlist);

delitem(nvlist, "data2");

print(nvlist);

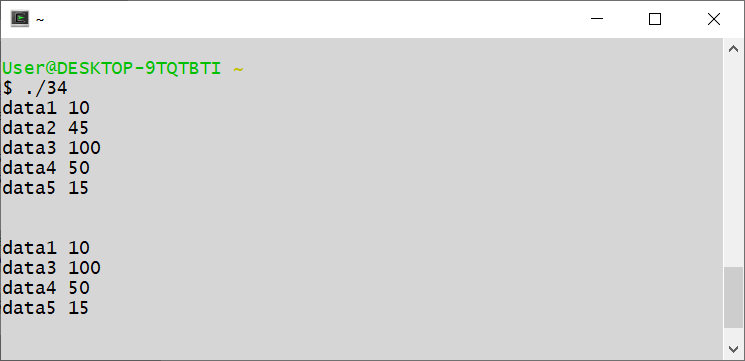
freeall(nvlist);

//print(nvlist);

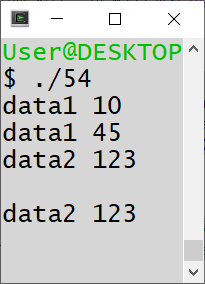
return 0;

}

**Результати роботи програми (скриншоти):**



**Результати роботи програми (скриншоти):**



**Завдання № 12:**

Напишите нетипизированное определение типа Type\_List на языке С. 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?

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

typedef struct node\_t node\_t;

struct node\_t {

void \*value;

node\_t \*next;

node\_t \*head;

};

node\_t\* make\_node(void \*value) {

node\_t \*node = malloc(sizeof(node\_t));

\*node = (node\_t) {.value = value, .next = NULL};

return node;

}

void push\_front(node\_t \*list, void \*value) {

node\_t \*temp = make\_node(value);

temp->next = list->head;

list->head = temp;

}

void push\_back(node\_t \*list, void \*value) {

node\_t \*temp = list->head;

for ( ; temp != NULL && temp->next != NULL; temp = temp->next)

continue;

if (temp != NULL) {

temp->next = make\_node(value);

} else {

list->head = make\_node(value);

}

}

typedef void (\*some\_func)(void\* );

void print\_string(void \*ptr) {

puts(\*(const char\*\*)ptr);

}

void free\_name(void \*ptr) {

free(\*(char\*\*)ptr);

}

void foreach(node\_t \*list, some\_func func) {

node\_t \*p;

for (p = list->head; p != NULL; p = p->next) {

func(p->value);

}

}

void clear(node\_t \*list, some\_func free\_name) {

foreach(list, free\_name);

node\_t \*p;

for (p = list->head; p != NULL; p = p->next) {

free(p);

}

free(list);

}

int main() {

node\_t \*list = malloc(sizeof(node\_t));

\*list = (node\_t) { .head = NULL };

char \*a = strcpy(malloc(strlen("abc") + 1), "abc");

char \*b = strcpy(malloc(strlen("def") + 1), "def");

char \*c = strcpy(malloc(strlen("ghi") + 1), "ghi");

char \*d = strcpy(malloc(strlen("jkl") + 1), "jkl");

push\_back(list, &a);

push\_back(list, &b);

push\_back(list, &d);

push\_back(list, &c);

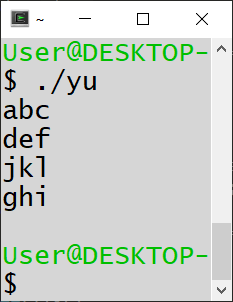
foreach(list, print\_string);

clear(list, free\_name);

// foreach(list, print\_string);

}

**Результати роботи програми (скриншоти):**



**Завдання № 13:**

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

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct List List;

struct List {

char \*name;

int value;

List \*next;

};

List \*lookup(List \*listp, char \*name){

for ( ; listp != NULL; listp = listp->next)

if (strcmp(name, listp->name) == 0)

return listp;

return NULL;

}

List \*create\_item(char \*name, int value){

List \*newp;

newp = (List \*)malloc(sizeof(List));

newp->name = name;

newp->value = value;

newp->next = NULL;

return newp;

}

List \*push\_front(List \*listp, List \*newp){

newp->next = listp;

return newp;

}

List \*push\_end(List \*listp, List \*newp){

List \*p;

if (listp == NULL)

return newp;

for (p = listp; p->next != NULL; p = p->next);

p->next = newp;

return listp;

}

void free\_list(List \*listp){

List \*next;

for ( ; listp != NULL; listp = next) {

next = listp->next;

/\* assumes name is freed elsewhere \*/

free(listp);

}

}

List \*del\_item(List \*listp, char \*name){

List \*p, \*prev;

prev = NULL;

for (p = listp; p != NULL; p = p->next) {

if (strcmp(name, p->name) == 0) {

if (prev == NULL)

listp = p->next;

else

prev->next = p->next;

free(p);

return listp;

}

prev = p;

}

return NULL;

}

void print(const List \*listp){

const List \*p;

for (p = listp; p != NULL; p = p->next) {

printf("%s: %d ", p->name, p->value);

}

}

List \*create\_item1(char \*name, int value) {

List \*newp;

newp = (List \*)malloc(sizeof(List));

newp->name = name;

newp->value = value;

return newp;

}

List \*copy(const List \*listp) {

if(listp == NULL)

return 0;

List \*newp = create\_item1(listp->name, listp->value);

newp->next = copy(listp->next);

return newp;

}

List \*merge(List \*lista, List \*listb){

List \*p;

if (lista == NULL)

return listb;

for (p = lista; p->next != NULL; p = p->next)

continue;

p->next = listb;

return lista;

}

List \*split(List \*listp, char \*name, List \*\*other){

List \*p, \*prev;

prev = NULL;

for (p = listp; p != NULL; p = p->next) {

if (strcmp(p->name, name) == 0) {

\*other = p;

if (prev == NULL)

listp = NULL;

else

prev->next = NULL;

return listp;

}

prev = p;

}

\*other = NULL;

return listp;

}

List \*insert\_before(List \*listp, List \*item, List \*newp){

List \*p, \*prev;

prev = NULL;

for (p = listp; p != NULL; p = p->next) {

if (p == item) {

p = push\_front(item, newp);

if (prev == NULL)

listp = p;

else

prev->next = p;

return listp;

}

prev = p;

}

return NULL;

}

List \*insert\_after(List \*listp, List \*item, List \*newp){

item->next = push\_front(item->next, newp);

return listp;

}

int test\_copy(List \*listp) {

List \*c;

int equal;

c = copy(listp);

equal = lists\_equal(listp, c);

free\_list(listp);

free\_list(c);

return equal;

}

int main(){

List \*listp = NULL, \*listp1 = NULL, \*merged = NULL;

List \*other;

printf("original\n");

listp = push\_front(listp, create\_item("A", 2));

listp = push\_front(listp, create\_item("B",8));

listp = push\_front(listp, create\_item("C", 5));

listp = push\_front(listp, create\_item("D", 4));

print(listp);printf("\n\n");

printf("insert\n");

listp = insert\_before(listp, listp, create\_item("E", 3));

listp = insert\_before(listp, lookup(listp, "C"), create\_item("G", 7));

insert\_after(listp, lookup(listp, "B"), create\_item("F", 6));

del\_item(listp, "B");

print(listp); printf("\n\n");

printf("copy\n");

listp1 = copy(listp);

print(listp1); printf("\n\n");

printf("split\n");

listp = split(listp, "C", &other);

print(listp); printf("\n\n");

printf("merge\n");

merged = merge(listp1, listp);

print(merged); printf("\n\n");

return 0;

}

/\*

int lists\_equal(const List \*lista, const List \*listb) {

for( ; lista != NULL && listb != NULL; lista->next, listb->next) {

if (lista->value != listb->value || strcmp(lista->name, listb->name) != 0)

return 0;

}

return lista == NULL && listb == NULL;

}

int test\_copy(List \*listp) {

List \*c = copy(listp);

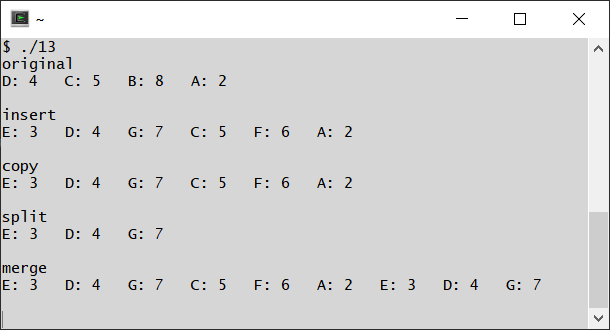
int equal = lists\_equal(listp, c);

return equal;

}

\*/

**Результати роботи програми (скриншоти):**



**Завдання № 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 O(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.

**Код програми:**

**a)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Tree Tree;

struct Tree {

char \*name;

int value;

Tree \*left;

Tree \*right;

};

int comparison(int brancha, int branchb){

if(brancha >= branchb)

return brancha;

else

return branchb;

}

int height(struct Tree \*treep){

if(treep == NULL)

return 0;

int l\_height = height(treep->left);

int r\_height = height(treep->right);

return comparison(l\_height, r\_height) + 1;

}

int check(struct Tree \*treep){

if(treep == NULL)

return 1;

int l\_height = height(treep->left);

int r\_height = height(treep->right);

if(abs(l\_height - r\_height) <= 1 && check(treep->right) &&

check(treep->left))

return 1;

else

return 0;

}

Tree \*insert(Tree \*treep, Tree \*new\_item){

if(treep == NULL)

return new\_item;

int cmp = strcmp(new\_item->name, treep->name);

if(cmp==0)

printf("insert: duplicate entry %s ignored", new\_item->name);

else if (cmp < 0)

treep -> left = insert(treep->left, new\_item);

else

treep -> right = insert(treep->right, new\_item);

return treep;

}

Tree \*search(Tree \*treep, char \*name){

if (treep == NULL)

return NULL;

int cmp = strcmp(name, treep->name);

if(cmp == 0)

return treep;

else if(cmp < 0)

return search(treep->left, name);

else

return search(treep->right, name);

}

void preorder(Tree \*treep) {

if (treep != NULL) {

printf("%s %d\n", treep->name, treep->value);

preorder(treep->left);

preorder(treep->right);

}

}

int main() {

Tree \*treep = NULL;

treep = insert(treep, &(struct Tree) {.name = "A", .value = 9});

treep = insert(treep, &(struct Tree) {.name = "B", .value = 2});

treep = insert(treep, &(struct Tree) {.name = "C", .value = 87});

treep = insert(treep, &(struct Tree) {.name = "D", .value = 98});

treep = insert(treep, &(struct Tree) {.name = "Y", .value = 78});

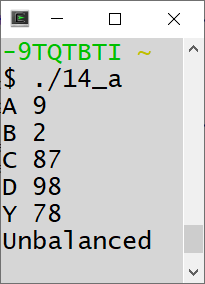
preorder(treep);

check(treep) ? printf("Balanced\n") : printf("Unbalanced\n");

return 0;

}

**Результати роботи програми (скриншоти):**



**b)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Tree Tree;

struct Tree {

char \*name;

int value;

Tree \*left;

Tree \*right;

};

int comparison(int brancha, int branchb) {

brancha >= branchb ? brancha : branchb;

}

int height(struct Tree \*treep){

if(treep == NULL)

return 0;

int l\_height = height(treep->left);

int r\_height = height(treep->right);

return comparison(l\_height, r\_height) + 1;

}

int check(struct Tree \*treep){

if(treep == NULL)

return 1;

int l\_height = height(treep->left);

int r\_height = height(treep->right);

if(abs(l\_height - r\_height) <= 1 && check(treep->right) && check(treep->left))

return 1;

else

return 0;

}

Tree \*insert(Tree \*treep, Tree \*new\_item){

if(treep == NULL)

return new\_item;

int cmp = strcmp(new\_item->name, treep->name);

if(cmp == 0)

printf("duplicate ignored");

else if (cmp < 0)

treep->left = insert(treep->left, new\_item);

else

treep->right = insert(treep->right, new\_item);

return treep;

}

void preorder(Tree \*treep) {

if (treep != NULL) {

printf("%s %d\n", treep->name, treep->value);

preorder(treep->left);

preorder(treep->right);

}

}

int main() {

Tree \*treep = NULL;

treep = insert(treep, &(struct Tree) {.name = "A", .value = 2});

treep = insert(treep, &(struct Tree) {.name = "B", .value = 4});

// treep = insert(treep, &(struct Tree) {.name = "C", .value = 6});

// treep = insert(treep, &(struct Tree) {.name = "D", .value = 8});

// treep = insert(treep, &(struct Tree) {.name = "E", .value = 10});

// treep = insert(treep, &(struct Tree) {.name = "F", .value = 12});

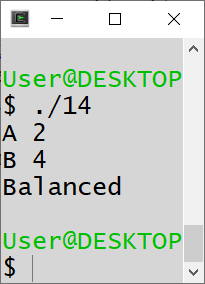
preorder(treep);

check(treep) ? printf("Balanced\n") : printf("Unbalanced\n");

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 15:**

**Код програми:**

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

#include<stdio.h>

#include <string.h>

#include <time.h>

static clock\_t clock\_start;

typedef struct Nameval Nameval;

struct Nameval {

char \*name;

int value;

Nameval \*left;

Nameval \*right;

};

void timer\_start(){

clock\_start = clock();

}

double timer\_elapsed(){

return ((double) clock() - clock\_start) / CLOCKS\_PER\_SEC;

}

Nameval \*insert(Nameval \*treep, Nameval \*newp){

if(treep == NULL)

return newp;

int cmp = strcmp(newp->name, treep->name);

if(cmp == 0)

printf("insert: dupcate entry %s ignored", newp->name);

else if (cmp < 0)

treep->left = insert(treep->left, newp);

else

treep->right = insert(treep->right, newp);

return treep;

}

Nameval \*lookup(Nameval \*treep, char \*name){

if(treep == NULL)

return NULL;

int cmp = strcmp(name, treep->name);

if(cmp == 0)

return treep;

else if(cmp < 0)

return lookup(treep->left, name);

else

return lookup(treep->right, name);

}

Nameval \*nrlookup(Nameval \*treep, char \*name){

while(treep != NULL){

int cmp = strcmp(name, treep->name);

if(cmp == 0)

return treep;

else if(cmp < 0)

treep = treep->left;

else

treep = treep->right;

}

return NULL;

}

int main(){

timer\_start();

Nameval \*treep = NULL;

struct Nameval newv = {.name = "Andy", .value = 12};

printf("%s %d\n", newv.name, newv.value);

treep = insert(treep, &newv);

treep = insert(treep, &(struct Nameval){.name = "Billy", .value = 18});

//printf("%g s\n", timer\_start());

Nameval \*res = lookup(treep, "Billy");

printf("%f s\n",timer\_elapsed());

printf("%s %d\n", res->name, res->value);

timer\_start();

res = nrlookup(treep, "Andy");

printf("%f s\n", timer\_elapsed());

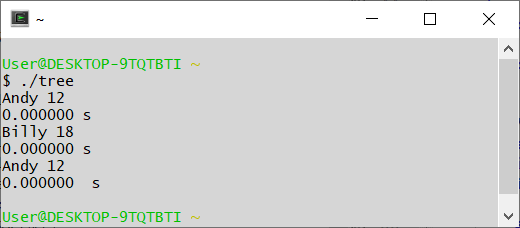
printf("%s %d\n", res->name, res->value);

printf("%f s\n", timer\_elapsed());

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 16:**

Напишите функцию сортировки с симметричным обходом (см. Unit 1 "Treetraverser") . 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?

**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Nameval Nameval;

struct Nameval {

char \*name;

int value;

Nameval \*left;

Nameval \*right;

};

Nameval \*insert(Nameval \*treep, Nameval \*newp){

if(treep == NULL)

return newp;

int cmp = strcmp(newp->name, treep->name);

if(cmp == 0)

printf("%s %d\n", newp->name, newp->value);

else if (cmp < 0)

treep->left = insert(treep->left, newp);

else

treep->right = insert(treep->right, newp);

return treep;

}

void print\_sorted(Nameval \*treep) {

if (treep == NULL)

return;

print\_sorted(treep->right);

printf("%s", treep->name);

print\_sorted(treep->left);

}

void inorder(Nameval \*treep) {

if (treep != NULL) {

inorder(treep->left);

printf("%s %d\n", treep->name, treep->value);

inorder(treep->right);

}

}

void preorder(Nameval \*treep) {

if (treep != NULL) {

printf("%s %d\n", treep->name, treep->value);

preorder(treep->left);

preorder(treep->right);

}

}

void postorder(Nameval \*treep) {

if (treep != NULL) {

postorder(treep->left);

postorder(treep->right);

printf("%s %d\n", treep->name, treep->value);

}

}

void applyinorder(Nameval \*treep, void (\*fn)(Nameval\*, void\*), void \*arg){

if (treep == NULL)

return;

applyinorder(treep->left, fn, arg);

(\*fn)(treep, arg);

applyinorder(treep->right, fn, arg);

}

void applypostorder(Nameval \*treep, void (\*fn)(Nameval\*, void\*), void \*arg){

if(treep == NULL)

return;

applypostorder(treep->left, fn, arg);

applypostorder(treep->right, fn, arg);

(\*fn)(treep, arg);

}

void printnv(Nameval \*p, void \*arg){

char \*fmt;

fmt = (char \*)arg;

printf(fmt, p->name, p->value);

}

int main() {

Nameval \*treep = NULL;

treep = insert(treep, &(struct Nameval){.name = "A", .value = 1});

treep = insert(treep, &(struct Nameval){.name = "F", .value = 6});

treep = insert(treep, &(struct Nameval){.name = "B", .value = 2});

treep = insert(treep, &(struct Nameval){.name = "E", .value = 5});

// treep = insert(treep, &(struct Nameval){.name = "C", .value = 3});

// treep = insert(treep, &(struct Nameval){.name = "D", .value = 4});

// treep = insert(treep, &(struct Nameval){.name = "G", .value = 7});

// treep = insert(treep, &(struct Nameval){.name = "I", .value = 8});

treep = insert(treep, &(struct Nameval){.name = "H", .value = 9});

// applyinorder(treep, printnv, "%s: %d\n");

// putchar('\n');

//applypostorder(treep, printnv, "%s: %d\n");

// printf("\n\n");

preorder(treep);

printf("\n\n");

inorder(treep);

printf("\n\n");

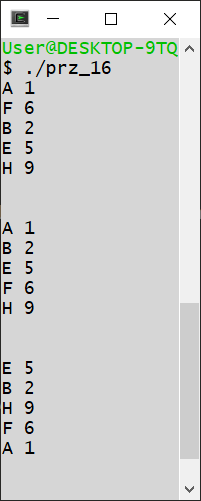
postorder(treep);

printf("\n\n");

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 17:**

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

**Код програми:**

**Результати роботи програми (скриншоти):**

**Завдання № 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.

**Код програми:**

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#define mod 89

enum{ MULTIPLIER = 31 };

typedef struct List List;

typedef unsigned char uch;

typedef unsigned int ui;

typedef struct List {

int data;

char \*string;

struct List \*next;

} List;

List \*symtab[mod];

ui hash(char \*str){

ui h = 0;

uch \*p;

for(p = (uch\*)str; \*p != '\0'; p++)

h = MULTIPLIER \* h + \*p;

return h % mod;

}

List \*search(char \*string, int create, int value){

List \*sym;

int h = hash(string);

for(sym = symtab[h]; sym != NULL; sym = sym->next)

if(strcmp(string, sym->string) == 0)

return sym;

if(create) {

sym = (List \*)malloc(sizeof(List));

sym->string = string;

sym->data = value;

sym->next = symtab[h];

symtab[h] = sym;

}

return sym;

}

void free\_list(List \* list){

List \*temp;

while(list){

temp = list;

list = list->next;

free(temp->string);

free(temp);

}

}

void show() {

List \*sym;

for (int i = 0; i < mod; ++i){

for(sym = symtab[i]; sym != NULL; sym = sym->next)

printf("%s %d\n", sym->string, sym->data);

}

}

int main(){

List \*node;

node = search("A", 1, 1);

node = search("B", 1, 2);

node = search("C", 1, 3);

node = search("D", 1, 4);

show();

//free\_list(node);

return 0;

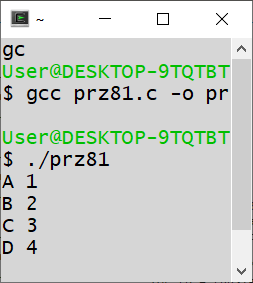
}

/\*Для того, щоб функція розприділяла дані в таблиці рівномірно найкраще

вибрати розмір масиву числом, яке мало б наймеше спільних дільників

при будь якому можливому значенні h \*/

**Результати роботи програми (скриншоти):**



**Завдання № 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.

Код програми:

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define NHASH 127

typedef struct Nameval Nameval;

typedef unsigned int ui;

typedef unsigned char uch;

struct Nameval {

char \*name;

int value;

Nameval \*next;

};

Nameval \*symtab[NHASH];

enum { MULTIPLIER = 31 };

ui hash(char \*str){

ui h = 0;

for(uch \*p = (uch \*)str; \*p != '\0'; ++p)

h = MULTIPLIER\*h + \*p;

return h % NHASH;

}

Nameval \*lookup(char \*name, int create, int value){

Nameval \*sym;

int h = hash(name);

for(sym = symtab[h]; sym != NULL; sym = sym->next)

if(strcmp(name, sym->name) == 0)

return sym;

if(create){

sym = (Nameval \*)malloc(sizeof(Nameval));

sym->name = name;

sym->value = value;

sym->next = symtab[h];

symtab[h] = sym;

}

return sym;

}

void create(){

lookup("a", 1, 1);

lookup("a0", 1, 2);

lookup("a000", 1, 3);

lookup("a0000", 1, 4);

lookup("a0000", 1, 5);

lookup("a0000", 1, 6);

lookup("a00000r", 1, 7);

}

void show() {

Nameval \*sym;

for(int i = 0; i < NHASH; ++i){

for (sym = symtab[i]; sym != NULL; sym = sym->next)

printf("%s %d\n", sym->name, sym->value);

}

}

int main(){

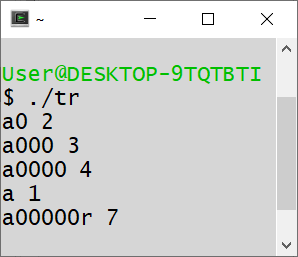
create();

show();

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 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?

**Код програми:**

#include <ctype.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

enum { MULTIPLIER = 31, NHASH = 127, SIZE = 6};

typedef struct Nameval Nameval;

typedef unsigned int ui;

struct Nameval {

char \*name;

int value;

Nameval \*next;

};

Nameval \*symtab[NHASH];

ui hash\_mod(char \*str, ui nhash){

ui h = 0;

unsigned char \*p;

for (p = (unsigned char \*) str; \*p != '\0'; p++)

h = MULTIPLIER \* h + \*p;

return h % nhash;

}

ui hash(char \*str) {

return hash\_mod(str, NHASH);

}

Nameval \*lookup(char \*name, int create, int value) {

Nameval \*sym;

int h = hash(name);

for (sym = symtab[h]; sym != NULL; sym = sym->next)

if (strcmp(name, sym->name) == 0)

return sym;

if (create) {

sym = (Nameval \*) malloc(sizeof(Nameval));

sym->name = name;

sym->value = value;

sym->next = symtab[h];

symtab[h] = sym;

}

return sym;

}

static char \*random\_string(char \*s, int len){

int i;

char c;

for (i = 0; i < len; i++) {

do {

c = rand() % 128;

} while (!isprint(c));

s[i] = c;

}

s[i] = '\0';

return s;

}

void create() {

lookup("a", 1, 1);

lookup("a0", 1, 2);

lookup("a000", 1, 3);

lookup("a0000", 1, 4);

lookup("a0000", 1, 5);

lookup("a0000", 1, 6);

lookup("a00000r", 1, 7);

}

static void nhash\_collisions() {

char string[SIZE + 1];

for (int i = 0; i < SIZE; ++i) {

random\_string(string, i);

string[i] = 'a';

string[i + 1] = '\0';

printf("%s %u\n", string, hash\_mod(string, MULTIPLIER));

}

}

void show() {

Nameval \*sym;

for(int i = 0; i < NHASH; ++i){

for (sym = symtab[i]; sym != NULL; sym = sym->next)

printf("%s %d %d \n", sym->name, sym->value, i);

}

}

int main() {

srand(time(NULL));

nhash\_collisions();

printf("\n");

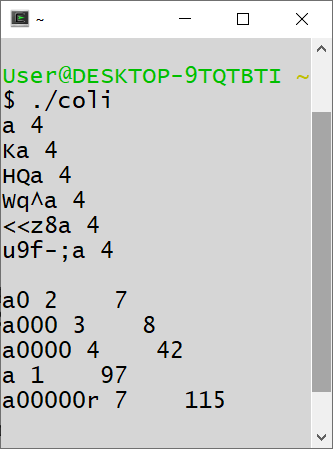
create();

show();

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 21:**

Напишите функцию для обращения к последовательным элементам хэштаблицы в несортированном порядке.

**Код програми:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define NHASH 11

typedef struct Nameval Nameval;

typedef unsigned int ui;

typedef unsigned char uch;

struct Nameval {

char \*name;

int value;

Nameval \*next;

};

Nameval \*symtab[NHASH];

enum { MULTIPLIER = 31 };

ui hash(char \*str){

ui h = 0;

for(uch \*p = (uch \*)str; \*p != '\0'; ++p)

h = MULTIPLIER\*h + \*p;

return h % NHASH;

}

Nameval \*lookup(char \*name, int create, int value){

Nameval \*sym;

int h = hash(name);

for(sym = symtab[h]; sym != NULL; sym = sym->next)

if(strcmp(name, sym->name) == 0)

return sym;

if(create){

sym = (Nameval \*)malloc(sizeof(Nameval));

sym->name = name;

sym->value = value;

sym->next = symtab[h];

symtab[h] = sym;

}

return sym;

}

void generate(){

lookup("Billy", 1, 12);

lookup("Alise", 1, 65);

lookup("John", 1, 6);

lookup("Endy", 1, 7);

lookup("Rozie", 1, 2);

lookup("Julia", 1, 3);

lookup("Martin", 1, 74);

lookup("Nick", 1, 0);

}

void show(){

Nameval \*sym;

for(int i = 0; i < NHASH; ++i){

for (sym = symtab[i]; sym != NULL; sym = sym->next)

printf("%s %d\n", sym->name, sym->value);

}

}

int main(){

generate();

show();

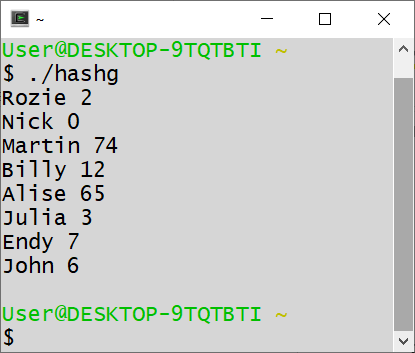
Nameval \*node = lookup("Billy", 1, 12);

printf("\n\n%s %d\n", node->name, node->value);

return 0;

}

**Результати роботи програми (скриншоти):**



**Завдання № 22:**

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

**Код програми:**

**Результати роботи програми (скриншоти):**

**Завдання № 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).

**Код програми:**

**a)**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

typedef struct {

doublex;

double y;

} point\_t;

enum { NHASH = 19, MULTIPLIER = 31 };

int hash\_point(point\_t point, int NHASH) {

return abs(point.x \*MULTIPLIER + point.y) % NHASH;

}

int main () {

srand(time(NULL));

int n;

scanf("%d", &n);

point\_t \*arr = malloc(n \* sizeof(point\_t));

for (int i = 0; i < n; ++i) {

arr[i].x = rand() % 128 \* (rand() % 2 ? 1 : -1);

arr[i].y = rand() % 128 \* (rand() % 2 ? 1 : -1);

//scanf("%d%d", &arr[i].x, &arr[i].y);

}

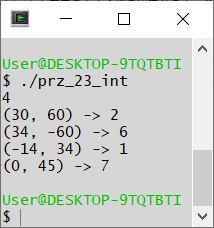
for (int i = 0; i < n; ++i) {

printf("(%lf, %lf) -> %d\n", arr[i].x, arr[i].y, hash\_point(arr[i], NHASH));

}

}

**Результати роботи програми (скриншоти):**



**Код програми:**

**b)**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef struct {

int x;

int y;

} point\_t;

enum { NHASH = 19, MULTIPLIER = 31 };

int hash\_point(point\_t point, int NHASH) {

return abs(point.x \*MULTIPLIER + point.y) % NHASH;

}

int main () {

srand(time(NULL));

int n;

scanf("%d", &n);

point\_t \*arr = malloc(n \* sizeof(point\_t));

for (int i = 0; i < n; ++i) {

arr[i].x = rand() % 64 \* (rand() % 2 ? 1 : -1);

arr[i].y = rand() % 64 \* (rand() % 2 ? 1 : -1);

//scanf("%d%d", &arr[i].x, &arr[i].y);

}

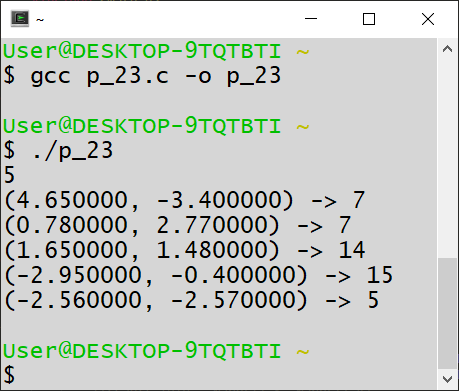
for (int i = 0; i < n; ++i) {

printf("(%d, %d) -> %d\n", arr[i].x, arr[i].y, hash\_point(arr[i], NHASH));

}

}

**Результати роботи програми (скриншоти):**



**Код програми:**

**c)**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef struct {

int x;

int y;

int z;

} point\_t;

enum { mod = 19, MULTIPLIER = 31 };

/\*

int hash\_point(point\_t point, int mod) {

return abs(point.x \*MULTIPLIER + point.y \* point.z) % mod;

}\*/

int hash\_point(point\_t point, int mod) {

return (point.x \* point.y + point.z ) % mod;

}

int main () {

srand(time(NULL));

int n;

scanf("%d", &n);

point\_t \*arr = malloc(n \* sizeof(point\_t));

for (int i = 0; i < n; ++i) {

arr[i].x = rand() % 64 \* (rand() % 2 ? 1 : -1);

arr[i].y = rand() % 64 \* (rand() % 2 ? 1 : -1);

arr[i].z = rand() % 64 \* (rand() % 2 ? 1 : -1);

//scanf("%d %d %d", &arr[i].x, &arr[i].y, &arr[i].z);

}

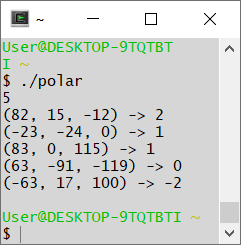
for (int i = 0; i < n; ++i) {

printf("(%d, %d, %d) -> %d\n", arr[i].x, arr[i].y, arr[i].z, hash\_point(arr[i], mod));

}

}

**Результати роботи програми (скриншоти):**



**Код програми:**

**d)**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

typedef struct {

int r;

int theta;

} point\_t;

enum { mod = 19, MULTIPLIER = 31 };

int hash\_point(point\_t point, int mod){

return ((point.r \* MULTIPLIER + point.theta + MULTIPLIER) \* MULTIPLIER) % mod;

}

int main () {

srand(time(NULL));

int n;

scanf("%d", &n);

point\_t \*arr = malloc(n \* sizeof(point\_t));

for (int i = 0; i < n; ++i) {

arr[i].r = cos(rand() % 2 \* (rand() % 2 ? 1 : -1));

arr[i].theta = tan(rand() % 1028 \* (rand() % 2 ? 1 : -1));

//scanf("%d%d", &arr[i].r, &arr[i].theta);

}

for (int i = 0; i < n; ++i) {

printf("(%d, %d) -> %d\n", arr[i].r, arr[i].theta, hash\_point(arr[i], mod));

}

}

**Результати роботи програми (скриншоти):**

