პროგრამირების პარადიგმები - შუალედური - 2024/25

ინსტრუქციები - აუცილებლად დაიცავით

ნაშრომის დატოვება

დესკტოპზე უნდა დაგხვდეთ დირექტორია რომლის სახელიც თქვენი უნივერსიტეტის ელ-ფოსტის პრეფიქსია. ამ დირექტორიაში უნდა დააკოპიროთ თქვენი ნაშრომები საგამოცდოს დატოვებამდე.
გირჩევთ თავიდანვე მოცემულ დირექტორიაში დააკოპიროთ საკითხები და მანდ იმუშაოთ ამოხსნებზე, რომ ბოლოს კოპირება არ დაგავიწყდეთ.

ტერმინალის გამოყენება

კოდის დასაწერად გირჩევთ გამოიყენოთ VSCode რედაქტორი რომელიც შეგიძლიათ იპოვოთ დესკტოპზე. ამოცანაზე სამუშაოდ მისი ფაილების გასახსნელად გამოიყენეთ: **File > Open Folder** და აირჩიეთ დესკტოპზე გადმოწერილი საგამოცდო საკითხების დირექტორია. კოდის დასაკომპილირებლად და ტესტების გასაშვებად შეგიძლიათ გამოიყენოთ VSCode-ში ჩაშენებული ტერმინალი: **Terminal > New Terminal**

ასევე შეგიძლიათ გამოიყენოთ Windows-ის სტანდარტული ტერმინალი. გახსენით საკითხის დირექტორია, **compress** მაგალითად, და ცარიელ ადგილას ჯერ დააჭირეთ **Control + Shift + Mouse-Right-Click** და შემდეგ აირჩიეთ **Open Command Prompt**

კოდის დაკომპილირებიდან მის გაშვებამდე კარგად დააკვირდით კომპილაცია წარმატებით დასრულდა თუ არა. კომპილაცია თუ ვერ შესრულდა, ძველი დაკომპილირებული პროგრამა (a.exe მაგალითად) უცვლელი ანუ ძველი რჩება. თუ გინდათ რომ დარწმუნდეთ ახალ დაკომპილირებულ პროგრამას ამოწმებთ გირჩევთ კომპილაციამდე ძველი კომპილაციის შედეგი წაშალოთ, მაგალითად ტერმინალში გაუშვით del a.exe

საკითხებთან ერთად მოცემული ტესტები არის მხოლოდ სამაგალითო, რათა გაგიადვილდეთ ნაშრომების შემოწმება. საბოლოო შეფასების დათვლისას ნაშრომები გასწორდება ტესტების სხვა სიმრავლეზე.

---

sum_3bit (20 ქულა)

თქვენი ამოცანაა იმპლემენტაცია გაუკეთოთ `sum_3bit` ფუნქციას, რომელმაც უნდა დააჯამოს მოცემული გამოსახულებების შედეგები.

თითოეული გამოსახულება აღწერილია ერთი ბაიტის გამოყენებით, ხოლო გამოსახულებების სიმრავლე მოცემული გაქვთ როგორც ბაიტების მიმდევრობა რომელიც ბოლოვდება ნულოვანი ბაიტით.

თითოეული ბაიტი აღწერს `x OP y` ტიპის გამოსახულებას, სადაც:

- ყველაზე დაბალი ინდექსის მქონე ორი ბიტი აღნიშნავს `OP`-ს
  - `00` აღნიშნავს `+` ოპერაციას
  - `01` აღნიშნავს `-` ოპერაციას
  - `10` აღნიშნავს `*` ოპერაციას
  - `11` აღნიშნავს `/` ოპერაციას (მთელი გაყოფა)
- მომდევნო სამი ბიტი `x`-ს
- და ბოლო სამი ბიტი `y`-ს

მაგალითად `11001000` აღწერს 6 + 2 გამოსახულებას, ხოლო `01001110` აღწერს 2 * 3 გამოსახულებას. მათი გაერთიანებით კი ვიღებთ: `11001000 01001110 00000000` == (6 + 2) + (2 * 3) = 14

ტესტირება

ნაშრომის დასაკომპილირებლად ტერმინალში გაუშვით: `gcc *.c` ხოლო ტესტებზე შესამოწმებლად: `.\a.exe`

---

process (ასემბლერი - 40 ქულა)

თქვენი ამოცანაა process.c ფაილში არსებული C პროგრამირების ენაზე დაწერილი კოდი გადათარგმნოთ კურსზე გავლილ მანქანურ/ასემბლი კოდში. თარგმანი ჩაწერეთ process.s ფაილში, სადაც უკვე მონიშნულია თუ სად უნდა დაწეროთ თარგმნილი კოდი.

process ფუნქციის სწორად თარგმნის შემთხვევაში აიღებს საკითხის 100%-ს.თუ process-ს ვერ მართავთ შეგიძლიათ გადათარგმნოთ მხოლოდ max ფუნქცია, რომელიც შეფასდება საკითხის 30%-ით.

თარგმნისას გაითვალისწინეთ რომ ქონვენშენების დაცვა საჭიროა როდესაც თქვენს კოდი გამოიძახებს სხვის ფუნქციას ან თქვენს კოდს გამოიძახებენ სხვა ფუნქციიდან. წინააღმდეგ შემთხვევაში ტესტები ჩაგეჭრებათ. ქონვენშენები რომლებიც უნდა დაიცვათ:

- ფუნქციის არგუმენტების სტეკზე ჩაწერა: ყველაზე მარცხენა არგუმენტი ჩაწერეთ ყველაზე ნაკლები ოფსეტის მისამართზე. მაგალითი: void fn(arg0, arg1) სტეკი: arg0: **sp**+0, arg1: **sp**+4... თქვენს დასაწერ ფუნქციაშიც არგუმენტები ასე დაგხვდებათ ჩაწერილი სტეკში.
- თქვენ მიერ დაწერილი ფუნქციებიდან დარეთარნდით, არ გამოიძახოთ exit().
- ფუნქციების return value შეინახეთ **x10** (იგივე **a0**) რეგისტრში. (ჩვენი ფუნქციის გამოძახება თუ დაგჭირდებათ ისიც **x10**-ში ჩაწერს return value-ს)
- ნებისმიერი ფუნქციის გამოძახებამ შეიძლება რეგისტრები გააფუჭოს ანუ ჩაწეროს სხვა მნიშვნელობები.
- სტრუქტურის field-ები მეხსიერებაში მიმდევრობითაა განლაგებული და გამოცდაზე შეგიძლიათ ჩათვალოთ, რომ მათ შორის padding-ები არ არის

ანუ struct {short a, int b} არის 6 ზომის სტრუქტურა, სადაც a არის შენახული addr+0ზე და b addr+2ზე. (alignment-ზე რაც ვილაპარაკეთ დაივიწყეთ)

- არ აქვს მნიშვნელობა ლოკალური ცვლადებისთვის რა თანმიმდევრობით გამოყოფთ სტეკზე მეხსიერებას. ასევე არ აქვს მნიშვნელობა სტეკზე რა ადგილას შეინახავთ **ra**-ს და ა.შ.

თუ თქვენი ამოხსნა ტესტებს გადის, ყველაფერი სწორია.

ტესტირება

თქვენი ამოხსნის გასაშვებათ ტერმინალში შეასრულეთ შემდეგი ბრძანება: `java -jar ../venus.jar process.s`

CountingArray (40 ქულა)

თქვენი ამოცანაა იმპლემენტაცია გაუკეთოთ CountingArray ჯენერიკ სტრუქტურას შემდეგი თვისებებით:

- ის ელემენტებს უნდა ინახავდეს ზრდადობის მიხედვით.
- ელემენტთან ერთად უნდა ინახავდეს თუ რამდენჯერ არის ის შენახული მასივში.
- ერთი და იგივე მნიშვნელობის რამოდენიმეჯერ დამატება უნდა იწვევდეს მხოლოდ მნიშვნელობის მთვლელის გაზრდას და მასივის სიგრძე უნდა რჩებოდეს უცვლელი.
- მნიშვნელობის წაშლსას მასივის ზომა უნდა შემცირდეს ერთით მხოლოდ მაშინ თუ მოცემული მნიშვნელობა მხოლოდ ერთხელ გვხვდებოდა მასივში.

მაგალითად თუ ცარიელ მასივში დავამატებთ მნიშვნელობებს შემდეგი მიმდევრობით: 1, 2, 1 მაშინ მასივის ზომა უნდა იყოს 2. ხოლო თუ შემდეგ ამოვშლით მნიშვნელობებს 1, 2 მაშინ მასივის ზომა უნდა გახდეს ერთი და მასში უნდა ინახებოდეს მხოლოდ ერთი 1-იანი.

CountingArray სტრუქტურას უნდა ქონდეს შემდეგი მეთოდები:

- void CountingArrayInit(CountingArray* a, int elem_size, CmpFn cmp_fn, FreeFn free_fn); - ინიციალიზაცია უნდა გაუკეთოს გადმოცემულ CountingArray ობიექტს. არგუმენტებად იღებს ელემენტების ზომას ბაიტებში, ელემენტების შედარების ფუნქციაზე მიმთითებელს, ელემენტების მეხსიერების გამათავისუფლებელ ფუნქციაზე მიმთითებლებს (თუ ასეთი საჭიროა).
- void CountingArrayDestroy(CountingArray* a); - უნდა გაათავისუფლოს გადმოცემული მასივის მიერ გამოყენებული მეხსიერება.
- int CountingArraySize(CountingArray* a); - უნდა დააბრუნოს მასივში შენახული **განსხვავებული** ელემენტების რაოდენობა.
- void* CountingArrayGet(CountingArray* a, int index); - უნდა დააბრუნოს მოცემულ ინდექსზე არსებული ელემენტის მისამართი. უკან **არ უნდა** დააბრუნოს მოცემული ელემენტის მეხსიერებაზე მფლობელობა.
- bool CountingArrayInsert(CountingArray* a, void* elem); - მასივში უნდა დაამატოს გადმოცემული ელემენტი და **აიღოს მის მეხსიერებაზე მფლობელობა (დუბლიკატების შემთხვევაშიც)**. გადმოცემული ელემენტები უნდა ინახებოდეს მათი მნიშვნელობების ზრდადობის მიმდევრობით. თუ დასამატებელი ელემენტი მასივში უკვე გვხვდება, მაშინ მასივის ზომა უცვლელი უნდა დარჩეს და მხოლოდ გადმოცემული ელემენტის მთვლელი უნდა გაიზარდოს.
- void CountingArrayRemove(CountingArray* a, void* elem); - მასივიდან უნდა წაშალოს მოცემული ელემენტის მხოლოდ ერთი მნიშვნელობა. მაგალითად თუ წასაშლელი ელემენტი მასივში ორჯერ არის შენახული, მაშინ მისი წაშლის შემდეგ მასივის ზომა უცვლელი უნდა დარჩეს ხოლო წასაშლელ მნიშვნელობაზე მასივმა უნდა შეამციროს მისი მთვლელი 2-დან 1-მდე. თუ გადმოცემული ელემენტი მასივში არ გვხვდება, მაშინ მისი წაშლის შემდეგ მასივი უცვლელი უნდა დარჩეს.
- void CountingArrayMerge(CountingArray* a, CountingArray* o); - o მასივის ყველა ელემენტი უნდა დაემატოს a მასივში. **მთლიანად იღებს მფლობელობას** o მასივზე და მის შიგთავსზე.

ტესტების 70%-ში ერთი და იგივე მნიშნველობის ელემენტი ორჯერ არ დაემატება მასივში.

CountingArrayMerge შეფასდება საკითხის შეფასების 10%-ით.

მეხსიერების არასწორად გამოყენების შემთხვევაში დაკარგავთ ტესტისთვის განკუთვნილი ქულის 15%-ს.

ნაკლები შეცდომები რომ დაუშვათ და კოდიც უფრო პატარა გამოგივიდეთ, გირჩევთ რომ მეპში გასაღბების და მნიშვნელობების მისამართების დათვლისთვის მაკროები (define) გამოიყენოთ.

ტესტირება

ნაშრომის დასაკომპილირებლად ტერმინალში გაუშვით: `gcc *.c` ხოლო ტესტებზე შესამოწმებლად: `.\a.exe`

<stdlib.h>

- void* malloc(size_t size); -- Allocates size bytes of uninitialized storage.
- void* realloc(void* ptr, size_t new_size); -- Reallocates the given area of memory. It must be previously allocated by malloc(), calloc() or realloc() and not yet freed with a call to free or realloc. Otherwise, the results are undefined.
- void free(void* ptr); -- Deallocates the space previously allocated by malloc(), calloc(), aligned_alloc, (since C11) or realloc().

<string.h>

- void* memcpy(void* dest, const void* src, size_t count); -- Copies count characters from the object pointed to by src to the object pointed to by dest. Both objects are interpreted as arrays of unsigned char.
- void* memmove(void* dest, const void* src, size_t count); -- Copies count characters from the object pointed to by src to the object pointed to by dest. Both objects are interpreted as arrays of unsigned char. The objects may overlap: copying takes place as if the characters were copied to a temporary character array and then the characters were copied from the array to dest. The behavior is undefined if access occurs beyond the end of the dest array. The behavior is undefined if either dest or src is a null pointer.
- char* strdup(const char* str1); -- Returns a pointer to a null-terminated byte string, which is a duplicate of the string pointed to by str1. The returned pointer must be passed to free to avoid a memory leak.
- char* strndup(const char* str, size_t size); -- Returns a pointer to a null-terminated byte string, which contains copies of at most size bytes from the string

pointed to by str. If the null terminator is not encountered in the first size bytes, it is added to the duplicated string.

- int strcmp (const char* str1, const char* str2); -- Compares the C string str1 to the C string str2. This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached. This function performs a binary comparison of the characters. For a function that takes into account locale-specific rules, see strcoll.
- int strncmp (const char* str1, const char* str2, size_t num); -- Compares up to num characters of the C string str1 to those of the C string str2. This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ, until a terminating null-character is reached, or until num characters match in both strings, whichever happens first.