1. Sortowanie

1 Założenia

Nie korzystamy z dynamicznego przydziału pamięci. To zagadnienie będzie omawiane na wykładzie w bliskiej przyszłości. Do tego czasu będziemy stosować "środek zastępczy" – definiowanie tablic o wystarczająco dużych rozmiarach.

2 Konstrukcja posortowanej tablicy elementów unikalnych z użyciem zmodyfikowanej funkcji `bsearch`

O zmodyfikowanej funkcji `bsearch`:

Oryginalna, biblioteczna funkcja `bsearch` zwraca adres znalezionego elementu tablicy albo – w przypadku braku szukanego elementu – `NULL`. Modyfikacja ma zastosowanie w sytuacji gdy szukany, ale nie znaleziony element ma być dodany do tablicy z zachowaniem obowiązującego w tablicy porządku. Funkcja `bsearch` „dociera" do informacji, pod jakim adresem „powinien" być szukany element (ale go tam nie ma). Jednak informacja ta jest tracona. Modyfikacja polega na przekazaniu (do funkcji wywołującej) adresu, pod którym należałoby umieścić szukany element.

Szablon programu należy uzupełnić o definicję funkcji `void *bsearch2(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *), char *result_adr)`. Możliwe są dwa rezultaty szukania:

1. Sukces szukania – funkcja wpisuje pod adres `result_adr` wartość różną od zera oraz zwraca adres znalezionego elementu (jak oryginalna funkcja `bsearch`).
2. Element z kluczem `*key` nie został znaleziony – funkcja wpisuje zero pod adres `result_adr` oraz zwraca adres, pod który należy wpisać nowy element zachowując przyjęty porządek. Funkcja nie sprawdza, czy zwracany adres nie przekracza zakresu pamięci przydzielonej tablicy `base`.

Zadanie to jest przykładem zastosowania funkcji `bsearch2`. Obejmuje wczytanie danych o artykułach spożywczych (cena jednostkowa, ilość, termin ważności[1] `dd.mm.yyyy`, nazwa) i zapisywanie ich w określonym porządku w tablicy struktur typu `Food`. Deklaracja tej struktury, zawierającej dane jednej partii artykułu, jest zapisana w szablonie.

Jeżeli wczytany rekord zawiera dane o artykule, który jest już zapisany w tablicy oraz dane wczytane różnią się (lub nie) od zapisanych tylko wartością w polu `amount`, to nie należy tworzyć dla tego rekordu nowego elementu tablicy, lecz zwiększyć wartość w polu `amount` istniejącej już struktury.

[1] "Termin ważności" należy traktować jako skrótowe określenie np. terminu przydatności do spożycia

Zapisywanie w tablicy struktur odczytanego ze strumienia wejściowego rekordu zawierającego dane o jednej partii artykułu ma zachowywać kolejność ustaloną wg kryteriów. Należy określić taką relację porządkującą elementy tablicy, aby odszukanie elementu, którego wartość pola `amount` ma być powiększona, wymagało tylko jednokrotnego wywołania funkcji `bsearch2`.

Kolejność elementów wg zadanego porządku ma być zachowana także w trakcie wpisywania nowych danych do tablicy.

Szablon programu należy uzupełnić o:

1. Definicję funkcji `bsearch2(...)` wg opisu powyżej.
2. Definicję funkcji `Food *add_record(Food *tab, int *np, ComparFp compar, Food *new)` , która wywołuje funkcję `bsearch2(...)` sprawdzającą, czy nowy artykuł (jego dane są zapisane pos adresem `*new`) jest zapisany w tablicy `tab` o `*np` elementach. O tym, czy uznać `*new` za nowy decyduje funkcja wskazywana pointerem do funkcji `compar` (typu `ComparFP` – zdefiniowanego w szablonie).
   - Jeżeli `*new` nie jest elementem nowym, to dane zapisane w elemencie tablicy są modyfikowane danymi zapisanymi w `*new` – konkretnie – ilość artykułu znalezionego w tablicy jest powiększana o ilość zapisaną w `*new`. Funkcja zwraca adres modyfikowanego elementu tablicy.
   - Jeżeli `*new` jest elementem nowym, to funkcja `add_record` dodaje we wskazanym miejscu nowy element (z ewentualnym przesunięciem części elementów tablicy), zwiększa liczbę elementów tablicy `*np` i zwraca adres wpisanego elementu.
3. Definicję funkcji wskazywanej pointerem `compar`.
4. Definicję funkcji `int read_goods(Food *tab, int no, FILE *stream, int sorted)`, która czyta `no` linii danych ze strumienia wejściowego. Dla każdego rekordu wywołuje funkcję `add_record`.

Test 1

Test wczytuje liczbę wprowadzanych linii danych, wywołuje funkcję `read_goods`, wczytuje nazwę artykułu i wypisuje wszystkie dane zawarte w strukturach z wskazaną nazwą artykułu (w kolejności: po pierwsze – rosnącej ceny, w drugiej kolejności – "rosnącego" terminu).

- **Wejście**
  1
  liczba linii n n linii: nazwa cena ilość dd.mm.yyyy
  nazwa artykułu
- **Wyjście**
  pamiętane w tablicy dane o artykule o wczytanej nazwie artykułu
  cena ilość dd.mm.yyyy
  cena ilość dd.mm.yyyy
  ...
- **Przykład**
  Wejście: 1
  6

kefir 3.50 30 7.6.2023
ser 7.80 25 15.6.2023
kefir 3.75 20 7.6.2023
ser 7.80 12 15.6.2023
mleko 3.25 44 29.12.2023
kefir 3.50 22 7.6.2023
kefir

Wyjście:
3.50 52.00 07.06.2023
3.75 20.00 07.06.2023

2.1 Sortowanie elementów tablicy struktur

Zadanie polega na posortowaniu biblioteczną funkcją `qsort` tablicy struktur utworzonej w zadaniu 8.1. Relację porządkującą elementy tablicy należy zdefiniować tak, aby przy możliwie małym koszcie obliczeniowym (dla dużej liczby danych) obliczyć wartość towaru, którego termin ważności mija dokładnie po n dniach od założonej daty (termin ważności = a dni + zadana data). Zadana data symuluje tu datę bieżącą.

Sugestia wyboru algorytmu: Posortować (qsort) wg daty, odszukać (bsearch) jeden element - w jego bezpośrednim "sąsiedztwie" są pozostałe z szukaną datą.

Należy zwrócić uwagę na okresy przełomu miesięcy lub lat. Zadanie można sobie ułatwić korzystając z funkcji deklarowanych w pliku nagłówkowym `time.h` standardowej biblioteki.

Szablon programu należy uzupełnić o:

1. Do odczytu danych można wykorzystać funkcję `read_goods()` z poprzedniego punktu z wartością parametru `sorted` = 0.
2. Definicję funkcji `float value(Food *food_tab, size_t n, Date curr_date, int days)`, która oblicza omawianą wyżej wartość artykułów.

Test 2

Test wczytuje dane tak, jak w zadaniu 1 (liczbę rekordów danych, rekordy danych, zadaną datę i liczbę dni do spradzanej daty ważności). Następnie wywołuje funkcję `value()`, która oblicza sumę wartości wszystkich artykułów, które tracą ważność za a dni (przykład: jeżeli 5.6.2023 będzie wczytaną datą, to będą poszukiwane artykuły o terminie ważności 10.6.2023).

- **Wejście**
  2
  liczba linii n
  n linii: nazwa cena ilość dd.mm.yyyy
  data (symulująca datę bieżącą) liczba a dni do szukanej daty ważności.
- **Wyjście**
  Suma wartości artykułów z wskazaną datą ważności
- **Przykład:**
  Wejście:
  2
  6

kefir 3.50 30 7.6.2023
ser 7.80 25 15.6.2023
kefir 3.75 20 7.6.2023
ser 7.80 12 15.6.2023
mleko 3.25 44 29.12.2023
kefir 3.50 22 7.6.2023
2 6 2023
5

Wyjście:
257.00

2.2 Linia sukcesji do brytjskiego tronu

Przydatne linki o zasadach sukcesji:
`https://www.royal.uk/encyclopedia/succession`
`https://pl.wikipedia.org/wiki/Linia_sukcesji_do_brytyjskiego_tronu`

Pełna lista sukcesji zawiera ponad 5700 osób. Dane o osobach (zapisane w szablonie programu) są ograniczone tylko do potomków Jerzego VI i samego Jerzego VI – przyjmijmy, że jest on pierwszym panującym, poniżej będzie nazywany *first*.

Założenia:

1. Przyjmujemy kolejność sukcesji obowiązującą po modyfikacji w roku 2013.
2. Dane o osobach (unikalne imię, płeć, data urodzenia, imię tego rodzica, po którym dziedziczona jest sukcesja) są pamiętane w tablicy struktur.
3. Kolejność elementów tablicy jest przypadkowa.
4. Imiona są unikalne.
5. *first* nie ma wpisanego imienia jego rodzica. W to miejsce jest wpisana wartość `NULL`.
6. Mimo ograniczenia liczby pretendentów, unikamy prostych algorytmów przeszukiwania całego zbioru danych (ze względu na dużą złożoność obliczeniową).
7. Z uwagi na (tymczasowy) brak możliwości dynamicznego przydziału pamięci, nie tworzymy drzewa genealogicznego.
8. Do zbioru pretendentów są dodane osoby, które już nie pretendują (bo panują, nie żyją albo abdykowali), ale ich dane są niezbędne do ustalenia kolejności sukcesji.

Jeżeli jest możliwość zdefiniowania funkcji, która dla dwóch dowolnych pretendentów wyznacza pierwszeństwo sukcesji na podstawie ich danych (zapisanych w nieuporządkowanym zbiorze), to należy taką funkcję zdefiniować i skorzystać z bibliotecznej funkcji `qsort`. I na tym można zakończyć to zadanie.

W przeciwnym przypadku należy zauważyć, że:

- Wielokrotnie powtarzaną operacją będzie szukanie w tablicy osoby spełniającej pewne warunki i przesunięcie jej danych do innego miejsca w tablicy.

- Czas szukania można skrócić wstępnie sortując elementy tablicy struktur z danymi o osobach.
- Aby zredukować liczbę przesunięć, należy dążyć do grupowania przesuwanych osób (aby przesuwać nie pojedynczy element tablicy, lecz większy, ciągły obszar pamięci).

Algorytm mógłby zawierać następujące etapy:

1. Sortowanie tablicy struktur z użyciem funkcji `qsort`.
   Funkcja porównująca dwa elementy tablicy powinna stosować kryterium takie, aby w posortowanej tablicy sąsiadowały ze sobą osoby, które prawdopodobnie będą blisko w kolejce sukcesji. Takie grupy to np. rodzeństwo (w kolejności wynikającej z zasady progenitury). W kryterium sortowania można też uwzględnić fakt, że pierwszym elementem tablicy ma być *first*.
2. Ustawianie rodzeństwa za ich rodzicem.

   Zaczynając od *first* należy znaleźć jego dzieci i przesunąć je bezpośrednio za *first*. Dalsza procedura jest chyba trywialna.

   Dla przyspieszenia odnajdywania dzieci jednego rodzica należy ten etap poprzedzić utworzeniem „tablicy indeksów".
3. Tworzenie tablicy indeksów.

   Każdy element "tablicy indeksów" powinien zawierać wskaźnik do imienia rodzica i indeks elementu tablicy osób, w którym są zapisane dane jego dziecka pretendującego w pierwszej kolejności (oraz ewentualnie liczbę dzieci).

   Jeżeli tablica osób będzie posortowana także według np. alfabetycznej kolejności rodzica, to do wyszukiwania rodzica w tablicy indeksów można użyć funkcji `bsearch` – to kryterium nie jest sprzeczne z wymaganiami wymienionymi w punkcie 1.
4. Usuwanie z kolejki osób, które nie są pretendentami.

   Tu jest dopuszczalne jednokrotne przeglądnięcie danych wszystkich osób zapisanych w tablicy `person_tab`.

Preferowany jest algorytm *in situ* (który nie przepisuje osób z jednej tablicy do innej, lecz do tej samej, ale w innym miejscu). Dopuszczalne jest korzystanie z tymczasowej tablicy wykorzystywanej do chwilowego pamiętania przesuwanych elementów.

Inne propozycje "oszczędnego" algorytmu mile widziane – proszę o nich poinformować prowadzącego zajęcia.

Typ struktury `Person` przewidzianej dla danych o pretendentach oraz typ struktury `Parent` są zdefiniowane w szablonie programu. Tablica osób `person_tab` jest zdefiniowana i inicjowana danymi w segmencie głównym programu.

Szablon programu należy uzupełnić o definicje funkcji:

- `int fill_indices_tab(Parent *idx_tab, Person *pers_tab, int size)`, która wypełnia "tablicę indeksów" typu `Parent` i zwraca liczbę wpisanych elementów tej tablicy;
- `void persons_shiftings(Person *person_tab, int size, Parent *idx_tab, int no_parents)`, która z wykorzystaniem funkcji `memcpy, memmove` przesuwa kolejne grupy elementów tablicy osób `person_tab`. Modyfikuje wartości indeksów w tablicy `idx_tab`;

- `int cleaning(Person *person_tab,int n)`, która usuwa z tablicy elementy osób niepretendujących (z wykorzystaniem jednej z funkcji `memcpy, memmove`). Funkcja zwraca liczbę pozostałych w tablicy pretendentów;
- `int create_list(Person *person_tab, int n)`, która wywołuje funkcję sortowania tablicy `qsort`, funkcje `fill_indices`, `persons_shiftings` oraz `cleaning`. . Zwraca liczbę pretendentów.

Należy także zdefiniować funkcje porównujące dla funkcji `qsort` i `bsearch`, zdefiniować tablicę indeksów (w takim miejscu, aby jej zasięg był możliwie mały).

Test 3

Test wywołuje funkcję `create_list`, wczytuje liczbę n i wypisuje imię pretendenta na n-tej pozycji w kolejce sukcesji.

- **Wejście**
  3 n
- **Wyjście**
  imię pretendenta
- **Przykład:**
  Wejście:
  3 24

  Wyjście
  David