Algorytmy i programowanie zajęcia 23 i 24

Marcin Żurowski

02 czerwca 2020

1 Stos i kolejka

Aby zbudować strukturę danych, która będzie bardziej skomplikowana niż zwykła zmienna czy tablica należy użyć konstrukcji która nazywa się **strukturą**.

1.1 Struktura

Strukturę w języku C przedstawimy na przykładzie punktu A umieszczonego w kartezjańskim układzie współrzędnych. Załóżmy, że mamy punkt A o współrzędnych $A = (2, 3.5)$ umieszczony na płaszczyźnie. Taki punkt możemy modelować za pomocą dwóch zmiennych:

```
double a_x = 2.0;
double a_y = 3.5;
```

Jednak jeśli mamy kilkaset takich punktów, można by ich współrzędne umieścić w tablicy. Jednak może łatwo dojść do pomyłki, która współrzędna to x a, która y, jeśli mamy bardziej skomplikowane elementy, jak na przykład punkty wraz z etykietami, które są dużą literą, to należało by użyć typu odpowiedniego dla znaków:

```
double a_x = 2.0;
double a_y = 3.5;
char a_label = 'A';
```

Oczywiście można użyć tutaj kodu ASCII i wtedy wszystkie dane możemy przechowywać w typie `double` z którego można zbudować tablicę, jednak przy bardzo skomplikowanych modelach może dojść do pomyłki. Nie wzięliśmy jeszcze pod uwagę napisów, które mogą nie przechowywać danych o stałej długości (wyrazy w słowniku są różnej długości). Z pomocą może przyjść tutaj konstrukcja zwana strukturą, którą definiujemy w następujący sposób:

```
struct punkt {
   double x;
   double y;
   char label;
};
```

W ten sposób tworzymy nowy typ którego zmienną definiujemy w następujący sposób:

```
struct punkt p;
```

Zmienna `p` zawiera trzy pola o następujących nazwach `x`, `y`, `label`. Aby odwołać się do odpowiednich pól np. po to aby nadać im odpowiednie wartości wykorzystujemy notację kropkową:

```
p.x = 2.0;
p.y = 3.5;
p.label = 'A';
```

Ponieważ zdefiniowana struktura tworzy nowy złożony typ, więc oczywiście możemy tworzyć z niego bardziej skomplikowane struktury jak tablica punktów:

```
struct punkt tab_p[10];
```

i do ósmego punktu w tablicy `tab_p` możemy do pola `x` wstawić wartość 0.5 w następujący sposób.

```
tab_p[7].x = 0.5;
```

Tak zdefiniowane przez nas zmienne mogą być polami w innych strukturach:

```
struct okrag {
   struct punkt o;
   double r;
};
```

na koniec tego podrozdziału postaramy się rozwiązać problem nazwy nowego typu, która składa się z dwóch wyrazów (`struct punkt`). W języku C znajduje się słowo kluczowe `typedef` za pomocą którego mozemy zmienić nazwę typu, który może mylić programistę np.:

```
typedef int numerek;
numerek a;
a = 4;
```

Warto tutaj zwrócić uwagę na składnie, która wygląda następująco:

```
typedef <definicja_starego_typu> <nowa_nazwa_typu>;
```

W ten sposób możemy zmienić nazwę naszej struktury w następujący sposób:

```
typedef
struct punkt {
   double x;
   double y;
   char label;
}
Punkt;
```

Proszę zauważyć, że użyłem nazwy typu rozpoczynająca się dużą literą, co ma zaznaczać, że `Punkt` jest typem złożonym (zawiera pola, co może przypominać paradygmat programowania obiektowego). Zmienną typu `Punkt` definiujemy w następujący sposób:

```
Punkt p;
```

1.2 Stos i kolejka

Termin wykonania poniższych zadań (23:59 08 czerwca 2020).

Zadanie polega na napisaniu Stosu i Kolejki opartych na tablicy przedstawionych w Wykładzie 8. Aby napisac stos musimy dokonać dwóch rzeczy:

- zdefiniować Stos będący strukturą danych o następujących polach:
  - `data` będąca dziesięcioelementową tablicą liczb całkowitych
  - `top` będące liczbą całkowitą wskazującą na szczyt stosu
- zdefiniować procedury i funkcję:
  - `init` procedura która inicjalizuje wszystkie pola w strukturze
  - `stack_empty` sprawdza czy stos jest pusty, i jeśli tak zwraca 1 w przeciwnym wypadku zwraca 0
  - `push` wkłada element na stos
  - `pop` zdejmuje element ze stosu i zwraca go
  - `write` wypisuje element podany jako argument (w programie należy użyć kilku takich funkcji)

Powyższe funkcje powinniśmy zadeklarować w następujący sposób:

```
void init(Stack * S);
int stack_empty(Stack S);
void push(Stack * S, int k);
int pop(Stack * S);
void write(Stack S);
void write_empty(int k);
void write_pop(int k);
```

Proszę zauważyć, że w niektórych deklaracjach wstawiamy stos przez wartość `Stack S` a w niektórych przypadkach przez wskaźnik `Stack * S`. Stos przekazujemy przez wskaźnik, wtedy, kiedy coś w nim zmieniamy (dodajemy element na stos, zdejmujemy element ze stosu), natomiast przez wartość wtedy kiedy nic nie zmieniamy (sprawdzamy czy stos jest pusty, wypisujemy stos). Kiedy przekazujemy stos przez wskaźnik do jego pól odwołujemy się w następujący sposób:

```
S->top = 1;
```

Po zaimplememntowaniu struktury i wszystkich funkcji w funkcji `main` wywołujemy zaimplementowane funkcje w następujący sposób:

```
STRUCT-STACK S
INIT(S) /*inicjuje zmienne*/
PUSH(S,1)
PUSH(S,2)
WRITE(S) //1 2
PUSH(S,3)
WRITE(S) //1 2 3
WRITE(STACK-EMPTY(S)) //false
WRITE(POP(S)) //3
WRITE(POP(S)) //2
WRITE(S) //1
WRITE(POP(S)) //1
WRITE(POP(S)) //underflow
WRITE(S) //
WRITE(STACK-EMPTY(S)) //true
```

W powyższym pseudokodzie po znaku `//` reprezentującym komentarz mamy wypisany tekst który powinien się znaleźć na standardowym wyjściu. Prawidłowy format wyjścia możemy sprawdzić uruchamiając program w zadaniu:
adjule zadanie: APR_047_STACK_AND_QUEUE

Kolejkę budujemy w podobny sposób jak stos. Najpierw definiujemy strukturę i procedury:

- zdefiniować Stos będący strukturą danych o następujących polach:
  - `data` będąca dziesięcioelementową tablicą liczb całkowitych
  - `length` będące liczbą całkowitą reprezentującą długość tablicy
  - `head` będące liczbą całkowitą reprezentującą indeks głowy kolejki
  - `tail` będące liczbą całkowitą reprezentującą indeks ogona kolejki
- zdefiniować procedury i funkcję:
  - `init` procedura która inicjalizuje wszystkie pola w strukturze
  - `queue_empty` sprawdza czy kolejka jest pusta, i jeśli tak zwraca 1 w przeciwnym wypadku zwraca 0
  - `enqueue` wkłada element do kolejki
  - `dequeue` zdejmuje element z kolejki i zwraca go
  - `write` wypisuje element podany jako argument (w programie należy użyć kilku takich funkcji)

Powyższe procedury deklarujemy według sposobu przedstawionego w stosie (argumenty których nie zmieniamy przekazujemy przez wartość, pozostałe argumenty przekazujemy przez wskaźnik) Po zaimplementowaniu struktury i wszystkich funkcji w funkcji `main` wywołujemy zaimplementowane funkcje w następujący sposób:

```
STRUCT-QUEUE Q
INIT(Q) /*inicjuje zmienne*/
ENQUEUE(Q,1)
ENQUEUE(Q,2)
WRITE(Q) //1 2
ENQUEUE(Q,3)
WRITE(Q) //1 2 3
WRITE(QUEUE-EMPTY(Q)) //false
WRITE(DEQUEUE(Q)) //1
WRITE(DEQUEUE(Q)) //2
WRITE(Q) //3
WRITE(DEQUEUE(Q)) //3
WRITE(DEQUEUE(Q)) //underflow
WRITE(Q) //
WRITE(QUEUE-EMPTY(Q)) //true
```

W powyższym pseudokodzie po znaku `//` reprezentującym komentarz mamy wypisany tekst który powinien się znaleźć na standardowym wyjściu. Prawidłowy format wyjścia możemy sprawdzić uruchamiając program w zadaniu:
adjule zadanie: APR_047_STACK_AND_QUEUE

1.3 Kopiec binarny i kolejka priorytetowa

Termin wykonania poniższych zadań (23:59 08 czerwca 2020).

O kopcu binarnym dowiedzieli się Państwo na Wykładzie 12, natomiast o kolejce priorytetowej mogą Państwo przeczytać [4]. Aby zaimplementować te dwie struktury, należy zdefiniować następująca strukturę i procedury

- zdefiniować Kopiec będący strukturą danych o następujących polach:
  - `data` będąca dziesięcioelementową tablicą liczb całkowitych
  - `length` będące liczbą całkowitą reprezentującą długość tablicy
  - `heap_size` będące liczbą całkowitą reprezentującą wielkość kopca
- zdefiniować procedury i funkcję:
  - `heap_init` procedura która inicjalizuje wszystkie pola w strukturze
  - `parent` zwraca indeks ojca
  - `left` zwraca indeks lewego syna
  - `right` zwraca indeks prawego syna
  - `max_heapify` po otrzymaniu indeksu węzła jako argumentu zamienia jego klucz z kluczami węzłów potomnych, o ile klucze te są większe
  - `build_max_heap` zmienia tablice w kopiec binarny
  - `heap_maximum` zwraca maksymalny element w kolejce

- `heap_extract_maximum` usuwa z kolejki największy element i zwraca go
- `heap_increase_key` zmienia klucz elementu o indeksie `arg1` na większy o wartości `arg2` i przywraca własność kopca
- `max_heap_insert` wstawia element do kolejki
- `write` wypisuje element podany jako argument (w programie należy użyć kilku takich funkcji)

Powyższe procedury deklarujemy według sposobu przedstawionego w stosie (argumenty których nie zmieniamy przekazujemy przez wartość, pozostałe argumenty przekazujemy przez wskaźnik) Po zaimplementowaniu struktury i wszystkich funkcji w funkcji `main` wywołujemy zaimplementowane funkcje w następujący sposób:

```
STRUCT-HEAP T
tab =(4,1,3,2,16,9,10,14,8,7)
HEAP-INIT(T,tab,tab.length)
BUILD-MAX-HEAP(T)
WRITE(T) //16 14 10 8 7 9 3 2 4 1

MAX-HEAP-INSERT(T,15)
WRITE(T) //16 15 10 8 14 9 3 2 4 1 7
WRITE(HEAP-MAXIMUM(T)) //16
WRITE(T) //16 15 10 8 14 9 3 2 4 1 7
WRITE(HEAP-EXTRACT-MAX(T)) //16
WRITE(T) //15 14 10 8 7 9 3 2 4 1
HEAP-INCREASE-KEY(T,4,19)
WRITE(T) //19 15 10 14 7 9 3 2 4 1
```

W powyższym pseudokodzie po znaku `//` reprezentującym komentarz mamy wypisany tekst który powinien się znaleźć na standardowym wyjściu. Prawidłowy format wyjścia możemy sprawdzić uruchamiając program w zadaniu:
adjule zadanie: APR_048_HEAP

Literatura

[1] B. Kernighan, D. Ritchie (2007). Język ANSI C. Wydawnictwo Naukowo-Techniczne.

[2] E. Palka (2012). Elementy algorytmiki dla początkujących. Wydawnictwo Naukowe UAM. (http://lib.amu.edu.pl/ksiazki-elektroniczne/)

[3] K. A. Ross, C. R. B. Wright (1996). Matematyka dyskretna. Wydawnictwo Naukowe PWN.

[4] T. H. Cormen, C. E. Leiserson, R. L. RIvest, C. Stein (2017). Wprowadzenie do algorytmów. Wydawnictwo Naukowe PWN.