Algorytmy i programowanie zajęcia 25 i 26

Marcin Żurowski

10 czerwca 2020

1 Lista z dowiązaniami

Aby skonstruować zaawansowaną strukturę jaką jest lista z dowiązaniami należy wprowadzić taki element programistyczny jakim jest:

1.1 Wskaźnik

1.1.1 Podstawy

Wskaźnikiem nazywamy zmienną, która może przechowywać adres innej zmiennej. Weźmy zmienną typu `Punkt` którego definicja wygląda następująco:

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

Rozmiar takiego typu to 16 bajtów na komputerach 64 bitowych czyli dwukrotna wielkość typu `double`. Wielkość typu i zmienne możemy sprawdzić za pomocą operatora `sizeof` którego używamy w następujący sposób:

```
Punkt p;
printf("%d %d\n", (int)sizeof(double), (int)sizeof p);
```

Do każdego zdefiniowanego typu możemy stworzyć typ wskaźnikowy za pomocą znaku `*`:

```
double * wd;
Punkt * wp;
```

Wartością każdego wskaźnika jest adres zmiennej na którą taki wskaźnik wskazuje. Na komputerach 64 bitowym taki adres ma 8 bajtów i dlatego wywołanie następującej instrukcji:

```
printf("%d %d\n", (int)sizeof wd, (int)sizeof wp);
```

uzyskamy dwie ósemki. Jeśli mamy zmienne wskaźnikowe, to chcielibyśmy aby te zmienne na coś wskazywały (przechowywały adres pewnych zmiennych). Aby uzyskać adres zmiennej należy użyć operatora unarnego `&` tak jak w przykładzie:

```
printf("%p\n", &p);
```

uzyskamy adres zmiennej `p` w systemie szesnastkowym. Jeżeli taki adres przypiszemy do wskaźnika za pomocą znanej instrukcji:

```
wp = &p;
```

to wskażemy tym wskaźnikiem na zmienną `p`. Teraz możemy do zmiennej, a w przypadku struktury do jej pól odwołać się na dwa sposoby:

- w tradycyjny sposób `p.x = 0.5;` czy odczyt zmiennej `printf("%d\n", p.x);`
- za pomocą wskaźnika używając unarnego operatora `*` w następujący sposób `(*p).y = 2.5;` czy odczyt zmiennej `printf("%d\n", (*p).y);`

nawias w wyrażeniu `(*p).y` jest niezbędny ponieważ operator `.` ma większy priorytet niż operator `*`, ale ponieważ wyrażenie to jest często używane zastąpiono je wyrażeniem `p->y`.

1.1.2 Wskaźniki, a dynamiczna alokacja pamięci

Zmienną każdego typu możemy powołać na dwa sposoby:

- za pomocą zmiennych automatycznych tworzy się zmienną `int a;` na stosie systemowym
  - **zalety:** pamięć jest automatycznie zwalniana w momencie zamknięcia bloku `}` w którym zmienną "powołano do życia"
  - **wady:** ograniczona wielkość stosu zależna od ustawień kompilatora
- dynamicznie za pomocą funkcji `malloc`. Funkcja `malloc` jest zadeklarowana w bibliotece `stdlib.h` w następujący sposób `void * malloc(size_t size);`. Funkcja ta jest wywoływana z jednym argumentem `size` typu `size_t`. Typ `size_t` to alias do typu `unsigned int`, czyli typ całkowity bez znaku, ponieważ za pomocą zmiennej `size` przekazujemy do funkcji wielkość bloku pamięci które chcemy zaalokować na stercie systemowej, liczoną w bajtach. Często rozmiar zmiennej danego typu jest różna na różnych komputerach, i aby zachować zgodność programu i za pomocą funkcji `malloc` zaalokować pamięć dla zmiennej na przykład typu `int` używamy operatora `sizeof` w następujący sposób: `malloc(sizeof(int));`. Funkcja `malloc` zwraca wskaźnik do zaalokowanej pamięci, jedna jest to wskaźnik do typu pustego `void *`, ponieważ `malloc` jest funkcja uniwersalną, którą można alokować zmienne każdego wcześniej zadeklarowanego typu. Aby zachować zgodność typów należy rzutować wynik zwracany przez tę funkcję na odpowiedni typ, i przechować go w zmiennej wskaźnikowej. Aby zaalokować dynamicznie zmienną typu `int` wywołujemy funkcję `malloc` w następujący sposób:

```
int * wa;
wa = (int *) malloc(sizeof(int));
```

- **zalety:** wielkość pamięci ograniczona przez wielkość fizyczną pamięci komputera
- **wady:** przed klamrą `}` zamykającą blok w którym została zdefiniowana zmienna wskaźnikowa `wa` przechowująca adres zaalokowanej pamięci należy zwolnić pamięć za pomocą funkcji `free` w następujący sposób:

  ```
  {
      int * wa;
      wa = (int *) malloc(sizeof(int));
      /* wiele innych instrukcji */
      free(wa);
  }
  ```

  Jeśli nie zwolnimy pamięci, po klamrze `}` zmienna automatyczna `wa` zostanie usunięta i stracimy wartość adresu pod którym pamięć została zaalokowana. Pamięc nie zwolniona będzie zablokowana i nie bedzie do niej dostępu. Jeśli powyższy blok jest ciałem jakiejś funkcji, a ta zostanie wywołana 100 razy, to po tym wykonaniu pamięć zużywana przez proces w którym jest uruchomiony nasz program zwiększy się o 800 bajtów! Jeżeli piszemy program, którego działanie jest długie, to po pewnym czasie "wycieki pamięci" spowodują niestabilność systemu (skończy się pamięć operacyjna, a taki proces zostanie zabity, aczkolwiek taka sytuacja występująca systemach Windows powoduje awarię systemu operacyjnego).

1.1.3 Przekazywanie wartości przez wskaźnik i przez wartość

Do dowolnej funkcji możemy przekazywać argument przez wartość albo przez wskaźnik. Ponieważ do funkcji możemy przekazywać wartości dowolnego typu, więc w przykładach skupię się na zmiennej typu `int`. Załóżmy, że chcemy napisać funkcję, która zwiększa zmienną o jeden, oto definicja takiej funkcji:

```
void f(int kopia_a)
{
    kopia_a++;
}
```

Kiedy wykonamy następujący ciąg instrukcji

```
int a = 4;
printf("%d\n", a);
f(a);
printf("%d\n", a);
```

otrzymamy jako wynik dwie czwórki! Dlaczego tak się dzieje? Otóż kiedy wywołujemy funkcję `f` tworzy się na stosie zmienna `kopia_a` do której zostaje skopiowana wartość zmienna `a`. W kolejnej linii `kopia_a` zostaje zwiększona o 1, natomiast zmienna `a` pozostaje bez zmian. Po zakończeniu funkcji `f` zmienna `kopia_a` zostaje zniszczona, natomiast zmienna `a` nigdy nie została nadpisana. Czy działanie naszej funkcji jest złe? Wprawdzie tak skonstruowana funkcja nie robi tego co powinna, ale możemy taką konstrukcje wykorzystać w innym celu. Na przykład jeśli musimy napisać funkcję, która nie powinna zmieniać wartości danych jako argument. Przykładem mogą być funkcję które mają wypisywać jakieś dane na standardowe wyjście.

Jednak, aby nasza funkcja działała poprawnie musimy przekazać naszą zmienną przez wskaźnik. Funkcję taką definiujemy w następujący sposób:

```
void f2(int * kopia_adresu_a)
{
    (*kopia_adresu_a)++;
}
```

Funkcje wywołujemy w następujący sposób:

```
int a = 4;
printf("%d\n", a);
f(&a);
printf("%d\n", a);
```

otrzymamy jako wynik czwórkę i piątkę, czyli taki wynik jakiego oczekiwaliśmy. A jak to działa? Otóż przekazujemy do funkcji adres zmiennej, a nie zmienną. Kiedy wywołujemy funkcję `f` tworzy się na stosie zmienna `kopia_adresu_a` do której zostaje skopiowana wartość adresu zmiennej `a`. W kolejnej linii odwołujemy się do zmiennej, na którą wskazuje wskaźnik `kopia_adresu_a` za pomocą operatora `*` i rzeczywista wartość zmiennej `a` zostaje zwiększona o 1, natomiast zmienna `kopia_adresu_a` w której jest przechowywany adres zmiennej `a` pozostaje bez zmian. Po zakończeniu funkcji `f` zmienna `kopia_adresu_a` zostaje zniszczona, natomiast zmienna `a` ma wartość 5.

1.2 Lista z dowiązaniami

Termin wykonania poniższych zadań (23:59 21 czerwca 2020).

O liście z dowiązaniami dowiedzieli się Państwo na Wykładzie 9. Aby zaimplementować tę strukturę danych, należy zdefiniować następujące struktury i procedury

- zdefiniować `Node` będący strukturą danych o następujących polach:
  - `key` będące liczbą całkowitą
  - `next` będące wskaźnikiem na `Node`
  - `prew` będące wskaźnikiem na `Node`

- zdefiniować `List` będącą strukturą danych o następujących polach:
  - `head` będące wskaźnikiem na `Node`
  - `tail` będące wskaźnikiem na `Node`
- zdefiniować procedury i funkcję:
  - `list_init` procedura która inicjalizuje wszystkie pola w strukturze `List`
  - `list_insert` wstawia element dany jako drugi argument do listy od strony głowy
  - `list_search` zwraca adres elementu, którego klucz jest dany jako drugi argument, a jeżeli na liście nie ma takiego elementu zwraca `NULL`
  - `list_insert_after` wstawia do listy element o kluczu danym jako drugi argument po elemencie o kluczu danym jako trzeci argument
  - `list_insert_before` wstawia do listy element o kluczu danym jako drugi argument przed elementem o kluczu danym jako trzeci argument
  - `list_delete` usuwa element z listy o kluczu danym jako drugi argument, lub nic nie robi jeśli nie ma elementu o takim kluczu na liście
  - `list_clear` zwalnia pamięć usuwając wszystkie `Node`
  - `list_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-LIST L
LIST-INIT(L) /*inicjuje zmienne*/
WRITE(L) //
LIST-INSERT(L,1)
LIST-INSERT(L,2)
WRITE(L) //2 1
LIST-INSERT(L,3)
WRITE(LIST-SEARCH(L,2)) //2
WRITE(LIST-SEARCH(L,4)) //-1
LIST-INSERT-AFTER(L,4,2)
WRITE(L) //3 2 4 1
LIST-DELETE(L,2)
WRITE(L) //3 4 1
```

```
LIST-DELETE(L,5)
WRITE(L) //3 4 1
LIST-INSERT-BEFORE(L,5,3)
WRITE(L) //5 3 4 1
LIST-CLEAR(L) /*zwalnia pamięć*/
WRITE(L) //
```

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_049_LIST

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.