Politechnika Śląska w Gliwicach
Wydział Automatyki, Elektroniki i Informatyki

Podstawy Programowania Komputerów

Plan

autor	Michał Skorus
prowadzący	dr inż. Agnieszka Danek
rok akademicki	2017/2018
kierunek	teleinformatyka
rodzaj studiów	SSI
semestr	1
termin laboratorium	czwartek, 10:00 – 11:30
grupa	2
sekcja	7
termin oddania sprawozdania	2018-01-25
data oddania sprawozdania	2018-01-25

1. Treść zadania

W pliku zawarte są dane zajęć w następującym formacie:

<godzina rozpoczęcia>-<godzina zakończenia> <dzień> <grupa> <prowadzący> <przedmiot>

Godzina jest podana w formacie: hh:mm, dzień przyjmuje wartości: pn, wt, sr, cz, pt, sb, nd. Grupa, prowadzący i przedmiot to pojedyncze wyrazy. Przykładowy plik:

08:30-10:00 pt gr1 Kowalski Programowanie
10:15-11:45 wt gr2 Nowak Fizyka
14:34-15:43 sr gr2 Kowalski Java
07:23-19:34 cz gr1 Nowak Astronomia

W wyniku działania programu powstają pliki dla każdego prowadzącego (nazwa pliku jest tożsama z nazwiskiem prowadzącego) zawierający plan zajęć dla prowadzącego. Kolejne
wpisu planu są posortowane chronologicznie. Przykładowy plik Kowalski.txt:

14:34-15:43 sr gr2 Java
08:30-10:00 pt gr1 Programowanie

Program uruchamiany jest z linii poleceń z wykorzystaniem następującego przełącznika: -i plik wejściowy

2. Analiza zadania

Zagadnienie przedstawia problem tworzenia planu lekcji dla każdego nauczyciela zawartego w pliku.

2.1. Struktury danych

W programie wykorzystano drzewo poszukiwań binarnych z nazwiskiem nauczyciela oraz podczepioną do tego drzewa listą jednokierunkową. W liście znajdują się dane potrzebne do pliku wyjściowego danego prowadzącego tj. godziny, dzień, grupa i przedmiot, które dodawane są w sposób sortujący listę rosnąco w zależności od dnia a później godziny. Węzeł drzewa może mieć od 0 do 2 potomków, przy czym po lewej stronie od węzła znajdują się potomkowie których zmienna prowadzący jest leksykograficznie mniejsza niż zmienna prowadzący węzła rodzicielskiego, po prawej zaś większe bądź równe. Taka struktura danych umożliwia łatwe posortowanie zarówno listy jak i drzewa.

2.2. Algorytmy

Program sortuje prowadzących poprzez umieszczenie je w drzewie binarnym. Wypisanie liczb posortowanych jest realizowane poprzez rekurencyjne przejście przez drzewo. Utworzenie drzewa i jego przejście jest wykonywane w średnim czasie O(n log n). W przypadku pesymistycznym (gdy drzewo zdegeneruje do listy) mamy O(n2). Natomiast w liście zastosowałem rozwiązanie iteracyjne, dane wstawiane są poprzez porównanie kolejno z każdy elementem listy która już jest posortowana, w najgorszym przypadku zajmuje to O(n) czasu. Jeżeli chodzi o złożoność pamięciową listy to O(2). Dzięki zastosowaniu takiej struktury danych program zachowuje balans między czasem a pamięcią jaki jest mu potrzebny.

3. Specyfikacja zewnętrzna

Program uruchamiany jest z linii poleceń. Należy przekazać do programu nazwę pliku wejściowego po odpowiednim przełączniku ”-i”, np.

program.exe -i plan.txt

Uruchomienie programu bez żadnego parametru lub z parametrem –h

program.exe
program.exe -h

powoduje wyświetlenie krótkiej pomocy. Uruchomienie programu z nieprawidłowymi parametrami powoduje wyświetlenie komunikatu:

```
Nieprawidlowe parametry
```

i wyświetlenie pomocy.

Podanie nieprawidłowej nazwy pliku powoduje wyświetlenie odpowiedniego komunikatu:

```
Plik wejsciowy < file_name > nie zostal znaleziony!
```

4. Specyfikacja wewnętrzna

Program został zrealizowany zgodnie z paradygmatem strukturalnym.

4.1. Typy zdefiniowane w programie

W programie zdefiniowano następujące typy:

```
enum week { pn = 1, wt = 2, sr = 3, cz = 4, pt = 5, sb = 6, nd = 7 };
```

Ten typ służy do przedstawienia dnia jako wartości dzięki której można później posortować listę wg dnia.

Zdefiniowano również typ służący do zbudowania listy danych.

```
struct list
{
	string time;
	week day;
	string group;
	string subject;
	list* next;
};
```

A także zdefiniowano typ drzewa binarnego.

```
struct node
{
	string teacher;
	list* head_data;
	node* lroot;
	node* rroot;
};
```

4.2. Ogólna struktura programu

W funkcji głównej wywoływana jest funkcja

```
bool LoadArguments(int argc, char ** argv, string &InFileName);
```

Funkcja ta sprawdza czy program został wywołany poprawnie i zapisuje w zmiennej `InFileName` Ścieżkę do pliku wejściowego. Funkcja główna zapisuje jej wynik w zmiennej **`bool load`**, jeśli wynik jest fałszem oznacza to złe parametry wywołania co wyświetla pomoc i kończy działanie programu, w przeciwnym wypadku program został wywołany poprawnie.

Następnie wywoływana jest funkcja:

```
void LoadFileInAndCheckFileIn(node* &tree, string &file_name);
```

Jej zadaniem jest otwarcie wywołanego wcześniej pliku wejściowego, sprawdzenie go, a następnie wpisanie jego danych do struktury, dzięki funkcjom :

```
void AddSortedLessons(node* &position, string &time, week &day, string &group,
string &subject);
```

i

```
void AddNewSortedNode(node* &tree, string &teacher, string &time, week day,
string &group, string &subject);
```

Dodają one nowe węzły lub nowe element listy w zależności czy funkcja `node* SearchExistTeacher(node* &tree, string &teacher);` zwróci jaką pozycje (`node* position`), czy też zwróci **`nullptr`**. Początkowa wartość tree wywoływana z funkcji main również wynosi **`nullptr.`**

Kolejnym krokiem działania programu jest funkcja `void CreateFilesInPreOrder(node* &tree);` przechodzi ona po każdym węźle (czyli każdym planie dla danego nauczyciela) w sposób PreOrder używając rekurencyjnie i tworzy dla niego plik wyjściowy (plan). Do utworzenia takiego pliku służy mu funkcja :

```
void CreateFile(node* &tree);
```

I na koniec funkcja main wywołuje funkcje `void DeleteDynamicStruct(node* &tree);` która całkowicie usuwa dynamicznie zaalokowaną strukturę pamięci, zaczynając od listy (funkcja `void DeleteList(list* &head);`), a potem liści drzewa. Po usunięciu drzewa funkcja zmienia wartość korzenia (`tree`) na **`nullptr`**.

4.3. Szczegółowy opis implementacji funkcji

```
bool LoadArguments(int argc, char ** argv, string &InFileName);
```

Funkcja sprawdza czy program został wywołany w poprawny sposób. Parametr `argc` przechowuje informacje ile argumentów zostało wywołane w konsoli, a parametr `argv` jest tablicą o `argc` elementach przechowującą wartości tych argumentów. `InFileName` to nazwa zmiennej do której zaraz zostanie przypisana ścieżka do pliku wejściowego. W funkcji **`int main()`** InFileName nosi nazwę **`file_name`**.

```
{
	const string INPUT("-i");
	const string HELP("-h");
	const int    InputFlag = 1;

	const int    correct = InputFlag;
	int result = 0;
```

```
	if (argc == 1)
	{
		Help();
		exit(6);
	}

	for (int i = 1; i < argc; i++)
	{
		string arg(argv[i]);
		if (arg == INPUT && argv[i+1])
		{
			InFileName = argv[i + 1];
			result |= InputFlag;
		}
		if (arg == HELP)
		{
			Help();
			exit(6);
		}
	}

	if (result == correct)
		return true;
	else
		return false;
}
```

Parametry:

INPUT, HELP – służą do porównania czy dany element tablicy nie jest przełącznikiem

result – przechowuje końcowy wynik

InputFlag – flaga przełącznika –i

correct – służy do porównania końcowego wyniku

Program zaczyna od sprawdzenia ilości argumentów podanych w konsoli. Jeżeli był tylko 1 wyświetla funkcje **Help()**, która wyświetla krótką instrukcję. Jeśli argumentów jest więcej pętla zaczyna zapisywać argument do zmiennej **arg**, jeżeli zgadza się z przełącznikiem –i oraz istnieje następny argument to zapisuje argument następny do `InFileName`. Następnie zmienia binarnie wartość **result** flagą **InputFlag**. Jeśli argument jest przełącznikiem –h to wyświetla instrukcje. Jeżeli wczytano ścieżkę funkcja zwraca true, w przeciwnym wypadku false.

Funkcja główna zapisuje wynik wyżej opisanej procedury do zmiennej **bool load**, a następnie sprawdza jej wynik czy parametry zostały dobrze wczytane i został użyty przełącznik. Później zostaje stworzony statycznie wskaźnik na korzeń drzewa poszukiwań binarnych, który ustawiany jest na **nullptr**.

```
bool LoadFileInAndCheckFileIn(node* &tree, string &file_name)

{
	fstream i(file_name, ios_base::in);
	if (!i.is_open()) {  cout << "\nPlik wejsciowy " << file_name << " nie
zostal znaleziony!" << endl; exit(1);}
	else { cout << "\nPlik wejsciowy zostal otworzony" << endl; }

	string time;
	string day_name;
	week day;
	string group;
	string teacher;
	string subject;
	string line;

	while (getline(i, line))
	{
```

Parametry:

time, day_name, day, group, teacher, subject, line – to parametry tymczasowe potrzebne do zczytania pliku

i – nazwa parametru pliku wejściowego

position – to wskaźnik na węzeł w drzewie zawierający istniejącego już prowadzącego, potrzebny do dopisania danych do listy

```
                        if (line == "")
                                continue;
                        istringstream str(line);
                str >> time >> day_name >> group >> teacher >> subject;
                day = convertday(day_name);
                node* position = SearchExistTeacher(tree, teacher);

                if (!CheckData(time, day_name, group, teacher, subject))
                {
                        cout << "\nBlednie wpisane dane w pliku"<<endl;
                        DeleteDynamicStruct(tree);
                        exit(7);
                }

                else if (!str)
                {
                        cout << "\nBlad danych, sprawdz zawartosc pliku wejsciowego"
<< endl;
                        DeleteDynamicStruct(tree);
                        exit(8);
                }

                else
                {
                        if (position == nullptr)
                                AddNewSortedNode(tree, teacher, time, day, group,
subject);
                        else
                        {
                                AddSortedLessons(position, time, day, group, subject);
                        }
                }
        }

        i.close();
        return true;
}
```

Funkcja **LoadFileInAndCheckFileIn()** otwiera plik wejściowy, pobiera z niego dane linijka po linijce. Jeżeli linijka jest pusta omija ją, jeżeli nie udało się stworzyć `istringstream str` program wyrzuca błąd, jeżeli dane zostały błędnie wpisane wyłapuje to **CheckData()**. Zanim jednak przejdzie do wpisywania do struktury wcześniej zamienia `string day_name;` na `enum week day;` przy pomocy `week convertday(string &day);` który też sprawdza poprawność dni wpisanych dni. Taka zamiana typów potrzebna jest do posortowania list wg dni i godzin. W programie jest również procedura `string convertNumber(int num);` która działa dokładnie tak samo jak **convertday()** tylko że zamienia w drugą stronę. Nie zostaną ona tutaj opisane, gdyż ich budowa jest trywialna, oparta na instrukcji warunkowej if(). Gdy program pobrał już i sprawdził daną linijkę sprawdza czy istnieje już taki prowadzący by w zależności od tego doczepić dane do listy wskazującej przez węzeł, czy też utworzyć nowy węzeł dla prowadzącego. Służy do tego funkcja:

```
node* SearchExistTeacher(node* &tree, string &teacher);

{
        node* tmp = tree;

        if (tmp == nullptr) return nullptr;
        else
```

```
	{
		if (teacher == tmp->teacher) return tmp;
		if (teacher < tmp->teacher)
		{
			SearchExistTeacher(tmp->lroot, teacher);
		}
		else
		{
			SearchExistTeacher(tmp->rroot, teacher);
		}
	}
}
```

Procedura przemieszcza się po drzewie w sposób rekurencyjny. Parametry jakie dostaje to korzeń drzewa `node* tree` oraz nazwa szukanego nauczyciela `string teacher`. Szukanie w drzewie poszukiwań binarnych jest bardzo optymalne podwzględem czasu, gdyż po jednej instrukcji warunkowej zawęża zakres poszukiwań o około połowę. Jeżeli korzeń wskazuje na `nullptr` to zwracany jest `nullptr`. Jeżeli szukany prowadzący ma taka samą nazwę jak prowadzący przypisany do aktualnie sprawdzanego węzła to funkcja zwraca pozycję – wskaźnik na węzeł. Jeżeli szukany prowadzący ma leksykograficznie mniejszą nazwę niż prowadzący przypisany do aktualnie sprawdzanego węzła to funkcja wywołuje samą siebie, ale podając za węzeł lewy korzeń. W każdym innym przypadku funkcja wywołuje samą siebie przechodząc do prawego korzenia.

Teraz gdy program wie już czy istnieje taki prowadzący w strukturze może wybrać czy dopisać się do listy, czy utworzyć węzeł.

Funkcja tworząca nowy korzeń:

```
void AddNewSortedNode(node* &tree, string &teacher, string &time, week day,
string &group, string &subject)

{
	if (!tree)
	{
		tree = new node{ teacher,new list{ time, day, group ,subject,
		nullptr },nullptr,nullptr };
	}
	else if (teacher < tree->teacher)
	{
		AddNewSortedNode(tree->lroot, teacher, time, day, group, subject);
	}
	else
	{
		AddNewSortedNode(tree->rroot, teacher, time, day, group, subject);
	}
}
```

Funkcja dostaje korzeń główny oraz wszystkie dane zczytane z pliku. Jeżeli korzeń wskazuje na `nullptr` to tworzony jest dynamicznie nowy węzeł z wpisanymi danymi wg struktury `struct node;`,tworząc tym samym pierwszy element listy. Następnie z racji że struktura poszukiwań binarnych jest posortowana oraz nowe dane są zawsze liśćmi zawartymi w danym przedziale to prowadzący również przez rekurencje szuka sobie miejsca w strukturze. Sposób przemieszczania jest identyczny jak w funkcji
`SearchExistTeacher();`

Jeśli jednak `SearchExistTeacher();` zwróciła pozycję inną niż `nullptr` to używana jest funkcja:

```
void AddSortedLessons(node* &position, string &time, week &day, string &group,
string &subject)

{

	//dodanie na poczatek
	if (position->head_data->day > day || (position->head_data->day == day &&
	position->head_data->time>time))
	{
		position->head_data = new list{ time, day, group, subject, position-
		>head_data };
	}

	//dodanie w środku
	else
	{
		list* help = nullptr;
		list* tmp = position->head_data;
		while (tmp)
		{
			if (tmp->day < day || (tmp->day == day && tmp->time<time))
			{
				if (tmp->next)
				{
					help = tmp;
					tmp = tmp->next;
				}
				else
					break;
			}
			if (tmp->day>day || (tmp->day == day && tmp->time >= time))
			{
				tmp = new list{ time, day, group, subject, tmp };
				help->next = tmp;
				break;
			}
		}

	//dodanie na koniec
		if (!tmp->next)
			tmp->next = new list{ time,day,group,subject,nullptr };
	}
}
```

Podobnie jak wcześniej opisywana funkcja **`AddSortedLessons();`** dostaje wszystkie dane potrzebne do listy bez prowadzącego, gdyż nie jest on już potrzebny do sortowania ani do struktury i jego zmienna jest przechowywana tylko w drzewie. Funkcja ta również w momencie gdy ma umieścić dane w strukturze umieszcza je w sposób posortowany ze względu na: w pierwszej kolejności dzień tygodnia, a w drugiej godzinę w danym dniu. Zmienna `time` jest typu `string` natomiast leksykograficznie porównanie będzie równoważne zwykłemu porównaniu liczb. Oba zbiory alfabetu i liczb uporządkowane są rosnąco w tablicy kodów ASCII, dlatego nie ma różnicy między tymi dwoma typami w tej sytuacji.

Po głębszym namyśle rozbiłem problem sortowania na trzy przypadki, w których elementy listy dodawane są na początek, w środku i na końcu. Dodając na początek wystarczy porownać z pierwszym elementem czy dzień jest mnieszy bądź równy, jeśli jest równy to czy godzina jest mniejsza. Jeżeli wszystko się zgadza to zamienić głowę listy na nowy element i ustawić jego

wskaźnik na zamieniany element.

Dodając do środka nasze dane muszą spełnić dokładnie ten sam warunek tyle, że w pętli do momentu gdy braknie następnych elementów. Gdy nie spełni tego warunku wystarczy zapamiętać aktualny wskaźnik, by później w razie spełnienia warunku zmienić jego wskaźnik na wstawiny element listy, a następnie przesunąć aktualnie sprawdzany element na wskaźnik następny.

Gdy zabraknie następnych elementów to znaczy, że element musi iść na koniec listy.

Funkcja wykorzystuje iterację, dlatego też potrzebna była dodatkowa zmienna **`list* tmp`** oraz zmienna **`list* help`** zapamietująca poprzedni element potrzebny później do złączenia listy.

Program wczytał dane do struktury czas utworzyć pliki i wstawić tam uporządkowane wg polecenia dane. Posłuży do tego funkcja:

```
void CreateFilesInPreOrder(node* &tree)
{
	if (tree)
	{
		CreateFilesInPreOrder(tree->lroot);
		CreateFile(tree);
		CreateFilesInPreOrder(tree->rroot);
	}
}
```

Działa w sposób prosty i podobny do poprzednich funkcji rekurencyjnych. Jeśli istnieje drzewo to przemieszcza się po drzewie w sposób PreOrder (LKP) i przechodzi po każdym węźle drzewa tworząc dla niego plik dzięki funkcji:

```
void CreateFile(node* &tree)
{
	fstream out(tree->teacher + ".txt", ios_base::out);
	if (!out.is_open()) { cout << "Blad otwarcia pliku"<<endl;
	DeleteDynamicStruct(tree); exit(4); }
	else { cout << "Stworzono nowy plik " << tree->teacher << ".txt dla
	prowadzacego" << endl; }
	list* tmp = tree->head_data;
	while (tmp)
	{
		out << tmp->time << " " << convertNumber(tmp->day) << " " << tmp-
		>group << " " << tmp->subject << endl;
		tmp = tmp->next;
	}
	out.clear();
	out.close();
}
```

Tworzy ona nowy plik wyjściowy o nazwie prowadzącego, zapisując dane do środka w odpowiedniej kolejności. Jest tu użyta wcześniej wspomniana funkcja **`convertNumber();`**

Ostatnim zadanie programu jest zwolnienie pamięci. Do tego procesu używa funkcji:

```
void DeleteDynamicStruct(node* &tree)
{
	if (tree)
	{
		DeleteDynamicStruct(tree->lroot);		//PostOrder
		DeleteDynamicStruct(tree->rroot);
		DeleteList(tree->head_data);
		delete tree;
```

```
                tree=nullptr;
        }
}
```

Zaczyna w kolejności PostOrder czyli od liści, wchodząc do listy danego węzła i dzięki funkcji:

```
void DeleteList(list* &head)
{
        list* tmp;
        while (head)
        {
                tmp = head;
                head = tmp->next;
                delete tmp;
        }
}
```

Zwalnianie pamięci listy zaczyna się od zapamietania pozycji głowy przesunięcie głowy na następny element i zwolnienie zapamiętanego elementu. Gdy usunie całą listę, usuwany jest węzeł i tak dalej aż zniknie całe drzewo. Korzeń główny drzewa z racji że był zaalokowany statycznie nie jest możliwy do usunięcia dlatego jego wartość jest ustawiana na `nullptr`.

5. Testowanie

Program został przetestowany na róznego rodzaju plikach. Pliki niepoprawne (zawierające inne oznacznia dni, nie zawierający wszystkich parametrów w linii pliku, niezgodne ze specyfikacja) powoduja zgłoszenie błedu. Złe parametry wywołania wyświetlają błąd. Plik pusty nie powoduje zgłoszenia błedu, ani też nie tworzy zbędnych pustych plików. Błąd wyświetla się również gdy plik nie zostanie otwarty. Program został sprawdzony pod kątem wycieków pamięci. Program jest zabezpieczony pod szerokim kątem.

6. Wnioski

Program choć początkowo wydawał się łatwy do stworzenia podczas pracy przysporzył mi wiele niespodzianek tj.np. niezauważone wycieki pamięci, gdzieś zaniechanie przekazania referencji, stworzenie takiej struktury pamięci złożonej z dwóch rodzajów struktur czy też sortowanie wg dwóch zmiennych. Mała zmiana kodu próbując rozwiącać dość prosty problem dodała mi kilku godzin pracy, ale mimo wszystko stworzenie tego programu nie tylko wzbogaciło moją wiedzę, ale też sprawiło mi zabawę.