8 Struktury dynamiczne

Dotychczas mieliśmy do czynienia ze statycznymi danymi, które po utworzeniu w pamięci nie zmieniały swojego rozmiaru. Na przykład tablica ma stałą liczbę elementów podobnie jak struktura (rekord) ma stałą liczbę pól. W tym rozdziale przedstawione zostaną dynamiczne struktury danych, które potrafią zmieniać swój rozmiar podczas działania programu. Można dopisywać do nich kolejne elementy i można je usuwać.

Zanim przedstawimy jak deklarować i jak operować na strukturach dynamicznych w języku C, trochę definicji podstawowych pojęć z nimi związanymi.

Będziemy utożsamiać dowolny typ T ze zbiorem jego wartości. Na przykład

$$int = \{0, 1, -1, 2, -2, 3, -3, \ldots\}.$$

Jeśli T jest dowolnym niepustym typem, to symbolem $L(T)$ oznaczać będziemy zbiór list o elementach będących wartościami typu T. Zbiór ten zdefiniowany jest następująco:

$$L(T) = \{nil_L\} \cup T \times L(T) \tag{8.1}$$

Symbol nil_L oznacza pustą listę (listę niezawierającą żadnego elementu).

Jeśli $l = \langle x, y\rangle \in L(T)$, to x nazywamy głową a y ogonem listy l. Każdą niepustą listę można rozłożyć na głowę i ogon.

Przykład listy o elementach $1, 2, 3$:

$$\langle 1, \langle 2, \langle 3, nil_L\rangle\rangle\rangle \in L(int).$$

Głową powyższej listy jest 1 a jej ogonem ogona jest lista $\langle 3, nil_L\rangle$.

Lista wartości typu T i lista list wartości typu T, to zupełnie różne byty.

Dla dowolnego niepustego typu T:

$$L(L(T)) \neq L(T).$$

Na przykład, dla $T = int$, prawdą jest, że $\langle nil_L, nil_L\rangle \in L(L(int))$ ale $\langle nil_L, nil_L\rangle \notin L(int)$.

Przekrój powyższych dwóch zbiorów ma tylko jeden wspólny element:

$$L(L(int)) \cap L(int) = \{nil_L\}.$$

Lista jest samopodobna, tzn. jej część - ogon - jest również listą. Często będziemy z tego korzystać gdyż jeśli trzeba będzie coś zrobić z całą listą, to wystarczy przetworzyć jej głowę a następnie powtórzyć to samo postępowanie na jej ogonie.

Jedną z charakteryzacji list jest ich długość.

Długość listy $x \in L(T)$ oznaczać będziemy symbolem $len(x)$, wyrażającym następującą funkcję:

$$len : L(T) \mapsto \mathcal{N}.$$

Funkcję długości możemy, korzystając z samopodobieństwa list, zdefiniować następująco z użyciem rekurencji (definicji odwołującej się do samej siebie):

$$len(x) = \begin{cases} 0 & \text{gdy } x = nil_L, \\ 1 + len(y) & \text{gdy } x = \langle v, y \rangle. \end{cases} \tag{8.2}$$

Na przykład:

$$\begin{aligned} len(\langle 1, \langle 2, \langle 3, nil_L \rangle \rangle \rangle) &= 1 + len(\langle 2, \langle 3, nil_L \rangle \rangle) \\ &= 1 + 1 + len(\langle 3, nil_L \rangle) \\ &= 1 + 1 + 1 + len(nil_L) \\ &= 1 + 1 + 1 + 0 \\ &= 3 \end{aligned}$$

Bardzo łatwo jest mając wartość $x \in T$ oraz listę $y \in L(T)$ stworzyć listę, w której x będzie głową a y ogonem. Wykonuje to operacja $cons(x, y)$:

$$cons : T \times L(T) \mapsto L(T),$$

$$cons(x, y) = \langle x, y \rangle \tag{8.3}$$

Można powiedzieć, że operacja $cons(x, y)$ dostawia w jednym kroku x na początek listy y.

Trochę bardziej skomplikowaną jest operacją jest dostawienie $x \in T$ na koniec listy $y \in L(T)$. Wymaga to przejścia całej listy y i zmiany listy nil_L za jej ostatnim elementem. Wykonuje to operacja $snoc(x, y)$ zdefiniowana następująco[1]:

$$snoc : T \times L(T) \mapsto L(T),$$

$$snoc(x, y) = \begin{cases} \langle x, nil_L \rangle & \text{gdy } y = nil_L, \\ \langle v, snoc(x, z) \rangle & \text{gdy } y = \langle v, z \rangle. \end{cases} \tag{8.4}$$

[1] Dziwna nazwa funkcji **snoc** jest po prostu odwróceniem nazwy **cons**.

Przykład obliczenia:

$$\begin{aligned} snoc(3, \langle 1, \langle 2, nil_L \rangle\rangle) &= \langle 1, snoc(3, \langle 2, nil_L \rangle)\rangle \\ &= \langle 1, \langle 2, snoc(3, nil_L)\rangle\rangle \\ &= \langle 1, \langle 2, \langle 3, nil_L \rangle\rangle\rangle \end{aligned}$$

Liczba operacji potrzebnych do wykonania operacji $snoc(x, y)$ jest liniową funkcją długości listy y.

Jeśli $x \in T$ a $y \in L(T)$, to w następujący sposób możemy sprawdzić czy wartość x jest elementem listy y:

$$el : T \times L(T) \mapsto bool,$$

$$el(x, y) = \begin{cases} false & \text{gdy } y = nil_L, \\ (x = v) \vee el(x, z) & \text{gdy } y = \langle v, z \rangle. \end{cases} \tag{8.5}$$

Sprawdźmy czy 2 jest elementem listy $\langle 1, \langle 2, \langle 3, nil_L \rangle\rangle\rangle$:

$$\begin{aligned} el(2, \langle 1, \langle 2, \langle 3, nil_L \rangle\rangle\rangle) &= (2 = 1) \vee el(2, \langle 2, \langle 3, nil_L \rangle\rangle) \\ &= (2 = 1) \vee (2 = 2) \vee el(2, \langle 3, nil_L \rangle) \\ &= (2 = 1) \vee (2 = 2) \vee (2 = 3) \vee el(2, nil_L) \\ &= (2 = 1) \vee (2 = 2) \vee (2 = 3) \vee false \\ &= true \end{aligned}$$

Dwie listy $x, y \in L(T)$ można połączyć w jedną listę ze zbioru $L(T)$ tworząc nową listę składającą się z elementów listy x, po których dopisano elementy listy y. Funkcję łączącą dwie listy x i y oznaczać będziemy symbolem $app(x, y)$:

$$app : L(T) \times L(T) \mapsto L(T),$$

$$app(x, y) = \begin{cases} y & \text{gdy } x = nil_L, \\ \langle v, app(z, y)\rangle & \text{gdy } x = \langle v, z \rangle. \end{cases} \tag{8.6}$$

Na przykład:

$$\begin{aligned} app(\langle 1, \langle 2, nil_L \rangle\rangle, \langle 3, \langle 4, nil_L \rangle\rangle) &= \langle 1, app(\langle 2, nil_L \rangle, \langle 3, \langle 4, nil_L \rangle\rangle)\rangle \\ &= \langle 1, \langle 2, app(nil_L, \langle 3, \langle 4, nil_L \rangle\rangle)\rangle\rangle \\ &= \langle 1, \langle 2, \langle 3, \langle 4, nil_L \rangle\rangle\rangle\rangle \end{aligned}$$

Liczba operacji potrzebnych do połączenia dwóch list x i y zależy liniowo od długości listy x ponieważ trzeba odszukać jej koniec i dostawić listę y za ostatnim elementem listy x.

Możemy tworzyć listy, które zawierają wszystkie wartości oryginalnej listy ale w odwróconej kolejności.

Operacja $rev(x)$ tworzy listę o odwróconej kolejności elementów niż na liście x:

$$rev : L(T) \mapsto L(T),$$

$$rev(x) = \begin{cases} nil_L & \text{gdy } x = nil_L, \\ snoc(v, rev(y)) & \text{gdy } x = \langle v, y \rangle. \end{cases} \tag{8.7}$$

Przykładowe obliczenia funkcji rev:

$$\begin{aligned} rev(\langle 1, \langle 2, \langle 3, nil_L\rangle\rangle\rangle) &= snoc(1, rev(\langle 2, \langle 3, nil_L\rangle\rangle)) \\ &= snoc(1, snoc(2, rev(\langle 3, nil_L\rangle))) \\ &= snoc(1, snoc(2, snoc(3, rev(nil_L)))) \\ &= snoc(1, snoc(2, snoc(3, nil_L))) \\ &= snoc(1, snoc(2, \langle 3, nil_L\rangle)) \\ &= snoc(1, \langle 3, snoc(2, nil_L)\rangle \\ &= snoc(1, \langle 3, \langle 2, nil_L\rangle\rangle) \\ &= \langle 3, snoc(1, \langle 2, nil_L\rangle)\rangle \\ &= \langle 3, \langle 2, snoc(1, nil_L)\rangle\rangle \\ &= \langle 3, \langle 2, \langle 1, nil_L\rangle\rangle\rangle \end{aligned}$$

Powyższy sposób odwracania elementów jest mało efektywny gdyż wymaga wykonania liczby operacji będącej funkcją kwadratową długości oryginalnej listy[2]. Na ćwiczeniach postaramy się wykonać to funkcją napisaną w języku C, która wykonywać będzie liniową liczbę operacji.

Na koniec omawiania wybranych operacji na listach przedstawimy operację $del(x, y)$, która usuwa z listy $y \in L(T)$ wszystkie wystąpienia wartości $x \in T$:

$$del : T \times L(T) \mapsto L(T),$$

$$del(x, y) = \begin{cases} nil_L & \text{gdy } y = nil_L, \\ del(x, z) & \text{gdy } y = \langle v, z\rangle \wedge x = v, \\ \langle v, del(x, z)\rangle & \text{gdy } y = \langle v, z\rangle \wedge x \neq v. \end{cases} \tag{8.8}$$

Przykład obliczenia:

$$\begin{aligned} del(2, \langle 1, \langle 2, \langle 3, \langle 2, nil_L\rangle\rangle\rangle\rangle) &= \langle 1, del(2, \langle 2, \langle 3, \langle 2, nil_L\rangle\rangle\rangle)\rangle \\ &= \langle 1, del(2, \langle 3, \langle 2, nil_L\rangle\rangle)\rangle \\ &= \langle 1, \langle 3, del(2, \langle 2, nil_L\rangle)\rangle\rangle \\ &= \langle 1, \langle 3, del(2, nil_L)\rangle\rangle \\ &= \langle 1, \langle 3, nil_L\rangle\rangle \end{aligned}$$

Omówione dotychczas listy miały, poza swoją prostotą, tę wadę, że często operowanie na nich wymagało przeglądania ich wszystkich elementów. Może to być bardzo nieefektywne dla długich list. Nawet uporządkowanie elementów na liście w porządku niemalejącym ich wartości niczego nie może przyspieszyć, bo

[2]Dopisujemy kolejno elementy na końcu listy pustej, na końcu listy jednoelementowej, na końcu listy dwuelementowej, na końcu listy trzyelementowej itd. a każde takie dopisanie wymaga liniowej liczby operacji w stosunku do długości listy na końcu której wstawiamy dany element.

aby dostać się do n-tego elementu listy trzeba przejść wszystkie wcześniejsze $n-1$ elementów.

Przedstawimy teraz strukturę, która pozwala pomijać duże jej fragmenty podczas przechodzenia po niej.

Strukturą tą są drzewa binarne. Symbolem $D(T)$ oznaczać będziemy zbiór wszystkich drzew binarnych przechowujących wartości typu T. Zbiór ten jest zdefiniowany następująco:

$$D(T) = \{nil_D\} \cup D(T) \times T \times D(T) \tag{8.9}$$

Symbol nil_D oznacza puste drzewo, które nie przechowuje żadnej wartości. Jeśli $t = \langle x, y, z\rangle \in D(T)$, to

y nazywamy korzeniem drzewa t,

x nazywamy lewym poddrzewem drzewa t,

z nazywamy prawym poddrzewem drzewa t.

Przykład drzewa o elementach $1, 2, 3$:

$$\langle\langle nil_D, 1, nil_D\rangle, 2, \langle nil_D, 3, nil_D\rangle\rangle \in D(int)$$

Korzeniem powyższego drzewa jest 2, jego lewe poddrzewo $\langle nil_D, 1, nil_D\rangle$ zawiera wartość 1 a jego prawe poddrzewo $\langle nil_D, 3, nil_D\rangle$ zawiera wartość 3.

Przetwarzanie drzewa rozpoczyna się od jego korzenia ale można przejść potem bezpośrednio do jego prawego poddrzewa, pomijając przeglądanie całego jego lewego poddrzewa.

Jeśli $\langle nil_D, x, nil_D\rangle \in D(T)$, to x nazywamy liściem (nie ma on żadnych elementów ani w lewem ani w prawym poddrzewie).

Funkcją $leaves(x)$ możemy wyznaczyć zbiór wszystkich liści w drzewie x:

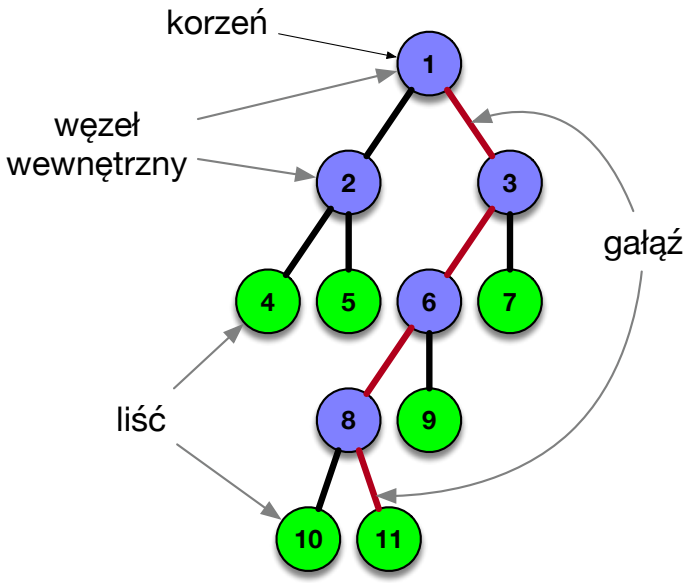
$$leaves : D(T) \mapsto 2^T,$$

$$leaves(x) = \begin{cases} \emptyset & \text{gdy } x = nil_D, \\ \{y\} & \text{gdy } x = \langle nil_D, y, nil_D\rangle, \\ leaves(y) \cup leaves(z) & \text{gdy } x = \langle y, v, z\rangle \wedge (y \neq nil_D \vee z \neq nil_D). \end{cases} \tag{8.10}$$

Przykład obliczeń:

$$\begin{aligned} leaves(\langle\langle nil_D, 1, nil_D\rangle, 2, \langle nil_D, 3, nil_D\rangle\rangle) &= leaves(\langle nil_D, 1, nil_D\rangle) \cup leaves(\langle nil_D, 3, nil_D\rangle) \\ &= \{1\} \cup \{3\} \\ &= \{1, 3\} \end{aligned}$$

Wszystkie węzły drzewa, które nie są liśćmi nazywamy węzłami wewnętrznymi. Każdy węzeł wewnętrzny ma przynajmniej jedną wartość w swoim lewym albo prawym poddrzewie.



Rysunek 8.1: Korzeń 1, węzły wewnętrzne $\{1, 2, 3, 6, 8\}$, liście $\{4, 5, 7, 9, 10, 11\}$ i wybrana gałąź $\langle 1, \langle 3, \langle 6, \langle 8, \langle 11, nil_L\rangle\rangle\rangle\rangle\rangle$ przykładowego drzewa.

Przechodząc od korzenia do liścia drzewa poruszamy się po ścieżce nazywanej gałęzią. Każdy liść wyznacza jedną gałąź rozpoczynającą się w korzeniu i kończącą się w tym liściu.

Gałąź drzewa będziemy reprezentować jako listę złożoną z wartości leżących na niej od korzenia do liścia.

Funkcja $branches(x)$ wylicza zbiór wszystkich gałęzi w drzewie x i jest zdefiniowana następująco:

$$branches : D(T) \mapsto 2^{L(T)},$$

$$branches(x) = \begin{cases} \emptyset & \text{gdy } x = nil_D, \\ \{\langle v, nil_L\rangle\} & \text{gdy } x = \langle nil_D, v, nil_D\rangle, \\ \bigcup_{b \in branches(y) \cup branches(z)} \{\langle v, b\rangle\} & \text{gdy } x = \langle y, v, z\rangle. \end{cases} \tag{8.11}$$

Przykład obliczenia:

$$branches(\langle\langle\langle nil_D, 4, nil_D\rangle, 2, nil_D\rangle, 1, \langle nil_D, 3, nil_D\rangle\rangle) = \{\langle 1, \langle 2, \langle 4, nil_L\rangle\rangle\rangle, \langle 1, \langle 3, nil_L\rangle\rangle\}$$

Na rysunku 8.1 przedstawiono przykładowe drzewo. Pod każdą wartością nie będącą liściem podpięto korzeń jej lewego i prawego poddrzewa.

Jedną z charakterystyk drzewa binarnego jest jego wysokość. Przez wysokość drzewa rozumiemy długość jego najdłuższej gałęzi.

Symbolem $h(x)$ będziemy oznaczać wysokość drzewa $x \in D(T)$. Wyliczyć można ją następująco:

$$h : D(T) \mapsto \mathcal{N},$$

$$h(x) = \max_{b \in branches(x)} len(b)$$

Powyższy wzór jest niepraktyczny ponieważ wymaga wyliczenia wszystkich gałęzi w drzewie, policzenia ich długości i wybrania maksimum z tych liczb.

W praktyce stosuje się następującą prostą definicję rekurencyjną:

$$h(x) = \begin{cases} 0 & \text{gdy } x = nil_D, \\ 1 + \max(h(y), h(z)) & \text{gdy } x = \langle y, v, z \rangle. \end{cases} \tag{8.12}$$

$$\begin{aligned} h(\langle\langle nil_D, 1, nil_D\rangle, 2, nil_D\rangle) &= 1 + \max(h(\langle nil_D, 1, nil_D\rangle), h(nil_D)) \\ &= 1 + \max(1 + \max(h(nil_D), h(nil_D)), h(nil_D)) \\ &= 1 + \max(1 + \max(0, 0), 0) \\ &= 1 + \max(1 + 0, 0) \\ &= 1 + \max(1, 0) \\ &= 2 \end{aligned}$$

Szczególną rolę odgrywają uporządkowane drzewa binarne. Spełniają one następujący predykat (warunek):

$$Ord(t) \equiv t = nil_D \vee (t = \langle x, v, y \rangle \wedge Ord(x) \wedge Ord(y) \wedge \forall_{z \in vals(x)}(z < v) \wedge \forall_{z \in vals(y)}(z > v)), \tag{8.13}$$

gdzie $vals(x)$ jest zbiorem wartości przechowywanych w drzewie $x \in D(T)$:

$$vals(x) = \begin{cases} \emptyset & \text{gdy } x = nil_D, \\ \{v\} \cup vals(y) \cup vals(z) & \text{gdy } x = \langle y, v, z \rangle. \end{cases} \tag{8.14}$$

Zwróć uwagę, że zgodnie z powyższą definicją, wartość w drzewie uporządkowanym występuje dokładnie jeden raz.

Drzewa o tej samej liczbie węzłów mogą mieć różną wysokość. Na przykład siedem węzłów mogą tworzyć drzewo wysokości 3 (patrz rysunek 8.2a) ale również drzewo wysokości 7 (patrz rysunek 8.2b).

Jeśli drzewo $x \in D(T)$ ma n węzłów, to jego wysokość h leży w zakresie:

$$\lceil \log_2(n+1) \rceil \leq h \leq n.$$

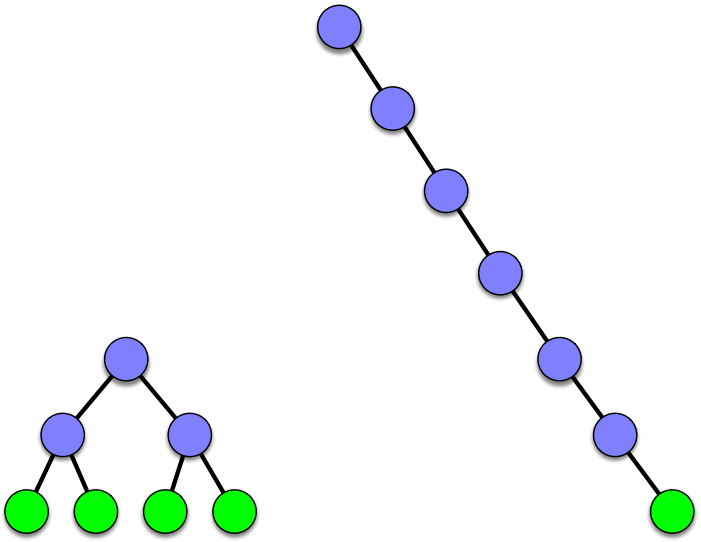
Na przykład, dla $n = 1\,023$, $10 \leq h(x) \leq 1\,023$.

W drugą stronę, liczba węzłów n drzewa $x \in D(T)$ o wysokości h, leży w zakresie:

$$h \leq n \leq 2^h - 1.$$

Na przykład, dla $h = 20$, $20 \leq n \leq 1\,048\,575$.

Drzewo z rysunku 8.2a nazywać będziemy zrównoważonym. W każdym jego poddrzewie jest tyle samo węzłów po lewej jak i po prawej stronie.



Rysunek 8.2: Przykłady drzew o siedmiu węzłach.

Drzewa zrównoważone spełniają następujący predykat:

$$Bal(t) \equiv t = nil_D \vee (t = \langle x, v, y\rangle \wedge Bal(x) \wedge Bal(y) \wedge |h(x) - h(y)| \leq 1). \quad (8.15)$$

W wielu zastosowaniach dąży się do tego aby drzewo pozostawało zrównoważone. Wymaga to często przerzucania węzłów między jego poddrzewami. Proces ten nazywa się równoważeniem drzewa.

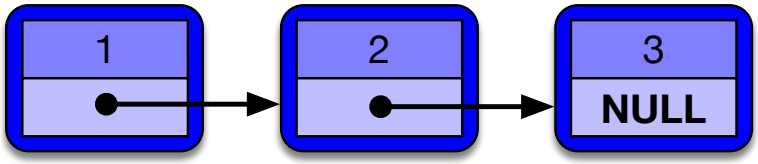
8.1 Listy jednokierunkowe

Typy, których wartości są krotkami wartości różnych typów reprezentujemy w języku C strukturami (rekordami). Zatem spróbujemy zadeklarować typ do reprezentowania list liczb całkowitych w postaci struktury `struct Lista` w następujący sposób:

```
// lista.c
struct Lista {
  int głowa;
  struct Lista ogon;
};
```

Próba kompilacji powyższej deklaracji kończy się błędem:

```
$ clang lista.c -o lista
lista.c:4:16: error: field has incomplete type 'struct Lista'
  struct Lista ogon;
               ^
```



Rysunek 8.3: Przykład listy złożonej z trzech elementów.

```
lista.c:2:8: note: definition of 'struct Lista' is not complete until the
      closing '}'
struct Lista {
       ^
1 error generated.
```

Błąd ten wynika z tego, że w chwili kompilacji pola `struct Lista ogon` nie jest znany rozmiar tej struktury (jej deklaracja jeszcze się nie zakończyła i być może zawiera ona jeszcze kolejne pola).

Taka „zapętlona” deklaracja struktury doprowadziłaby do stworzenia danej `struct Lista x` o nieskończonej liczbie pól `int głowa`:

```
x.głowa, x.ogon.głowa, x.ogon.ogon.głowa, x.ogon.ogon.ogon.głowa, ...
```

Do zadeklarowania listy konieczne jest skorzystanie z typu wskaźnikowego. Zamiast umieszczać w polu struktury całą tę strukturę, umieszczać będziemy w nim wskaźnik wskazujący taką strukturę (patrz rysunek 8.3).

Na listingu 8.1 przedstawiono plik nagłówkowy definiujący typ `Lista` jako wskazanie na strukturę `struct Węzeł` o dwóch polach:

`wartość` pole przechowujące wartość będącą liczbą całkowitą (głowa listy),

`następny` pole przechowujące wskazanie na strukturę przechowującą następny element listy (wskazanie na ogon listy).

Listing 8.1: Deklaracje list

```
// listy.h

struct Węzeł
{
  int wartość;
  struct Węzeł* następny;
};

typedef struct Węzeł* Lista;

void drukuj(Lista wsk);
Lista dopisz_na_poczatek(Lista wsk, int i);
void dopisz_na_koniec(Lista* wsk, int i);
void usun(Lista* wsk);
```

Definicja typu `Lista` upraszcza program, bo nie trzeba w wielu miejscach wpisywać pełnego typu `struct Węzeł*`.

Lista będzie reprezentowana połączonymi wskaźnikami węzłami, przy czym każdy węzeł przechowuje wartość będącą liczbą całkowitą (głowę listy) oraz wskazanie na następny węzeł (ogon listy).

Każdy węzeł zna jedynie wskazanie na następny węzeł, zatem można poruszać się po tej liście tylko w kierunku od jej początku ku końcowi i zawsze przechodzimy o jeden element (stąd nazwa *lista jednokierunkowa*).

W pliku nagłówkowym zadeklarowano również cztery funkcje jakie będą użyte w przykładzie:

`drukuj()` drukuje wartości przechowywane na liście,

`dopisz_na_poczatek()` dopisuje kolejną wartość na początku listy,

`dopisz_na_koniec()` dopisuje kolejną wartość na końcu listy,

`usun()` usuwa całą listę z pamięci.

8.1.1 Wstawanie na początek

Rozpoczniemy od operacji dopisującej kolejną wartość na początek listy. Jak wspomniano przy definicji operacji *cons* (patrz wzór (8.3)) wstawienie takie jest bardzo proste, gdyż wystarczy utworzyć nową listę o głowie takiej jak wstawiana wartość a ogonie takim jak lista, do której wstawiamy.

Na listingu 8.2 przedstawiono definicję funkcji `Lista dopisz_na_poczatek(Lista wsk, int i)`, której wartością jest wskazanie na listę do której dopisano na początku wartość całkowitą.

Listing 8.2: Wstawianie na początek

```
// dopisz_na_poczatek.c

#include <stdlib.h>
#include "listy.h"

Lista dopisz_na_poczatek(Lista wsk, int i)
{
  Lista nowy;
  nowy = malloc(sizeof(struct Węzeł));
  nowy->wartość = i;
  nowy->następny = wsk;
  return nowy;
}
```

Funkcja ta dostaje parametrem `wsk` wskaźnik na początek oryginalnej listy. Następnie rezerwuje pamięć na nowy węzeł listy (wywołując funkcję `malloc()`) i wypełnia jego pola wartościami `i` oraz `wsk`. Na koniec zwraca wskazanie na nowoutworzony węzeł.

Jeśli przechowujemy w zmiennej `p` typu `Lista` wskazanie na listę i chcemy dopisać na jej początek wartość 1, to wystarczy wywołać funkcję w następujący sposób:

```
p = dopisz_na_poczatek(p, 1);
```

Kolejne liczby można dopisać wywołując:

```
p = dopisz_na_poczatek(dopisz_na_poczatek(p, 2), 3);
```

8.1.2 Wstawianie na koniec

Jak wspomniano przy definicji operacji *snoc* (patrz wzór (8.4)), operacja wstawiania na koniec listy jest bardziej skomplikowana gdyż wymaga odnalezienia ostatniego elementu listy i dopisanie za nim kolejnej wartości.

Na listingu 8.3 przedstawiono definicję funkcji `dopisz_na_koniec(Lista* wsk, int i)`.

Listing 8.3: Wstawianie na koniec

```
// dopisz_na_koniec.c

#include <stdlib.h>
#include "listy.h"

void dopisz_na_koniec(Lista* wsk, int i)
{
  Lista nowy, pomoc;
  nowy = malloc(sizeof(struct Węzeł));
  nowy->wartość = i;
  nowy->następny = NULL;
  if(*wsk == NULL)
    *wsk = nowy;
  else
  {
    pomoc = *wsk;
    while(pomoc->następny != NULL)
      pomoc = pomoc->następny;
    pomoc->następny = nowy;
  }
}
```


Pierwsza widoczna różnica w porównaniu z funkcją `dopisz_na_poczatek()`, to że parametrem `wsk` funkcja nie dostaje wskazania na listę ale wskazanie na miejsce, gdzie znajduje się wskazanie na listę (wskazanie na wskazanie).

Dzięki temu, że parametr `wsk` wskazuje na miejsce gdzie znajduje się wskazanie na początek listy, funkcja `dopisz_na_koniec()` może zmienić to wskazanie gdy wartość będzie wstawiana na listę pustą (wskazanie zmieni się z NULL na adres nowoutworzonego elementu).

8.1.3 Drukowanie

Funkcja `drukuj(Lista wsk)`, przedstawiona na listingu 8.4, drukuje wartości przechowywane na liście, przy czym parametr `wsk` wskazuje na początek drukowanej listy (jej pierwszy węzeł).

Listing 8.4: Drukowanie listy

```
#include <stdio.h>
#include <stdlib.h>
#include "listy.h"

void drukuj(Lista wsk)
{
  while(wsk != NULL)
  {
    printf("%d ", wsk->wartość);
    wsk = wsk->następny;
  }
  printf("\n");
}
```

W pętli **while**, po każdym wydrukowaniu wartości `wsk->wartość`, wskaźnik `wsk` zostaje zmieniony na wartość wskaźnika z pola `następny` w strukturze wskazywanej tym wskaźnikiem. Dzięki temu nastąpi przejście do drukowania kolejnego elementu.

8.1.4 Usuwanie listy

Na listingu 8.5 przedstawiono funkcję `usun(Lista* wsk)`, która usuwa wszystkie elementy z listy wskazywanej przez `*wsk` (`wsk` jest wskazaniem na miejsce gdzie jest wskazanie na początek usuwanej listy).

Listing 8.5: Usuwanie listy

```
// usun.c

#include <stdlib.h>
#include "listy.h"

void usun(Lista* wsk)
{
  Lista pomoc;
  while(*wsk != NULL)
    {
      pomoc = (*wsk)->następny;
      free(*wsk);
      *wsk = pomoc;
    }
}
```

Dzięki temu, że `wsk` jest wskazaniem na wskazanie na początek listy, funkcja `usun()` może poprawnie umieścić we wskaźniku na początek listy wartość NULL (nie ma już tej listy).

W pętli **while** wartość `*wsk` wskazuje na kolejny węzeł do usunięcia. Zanim zostanie on usunięty, pod zmienną `pomoc` podstawiany jest wskaźnik z pola `następny` struktury wskazywanej przez `*wsk`. Zwróć uwagę na konieczność użycia nawiasów w wyrażeniu `(*wsk)->następny`. To dla tego, że operator `->` ma wyższy priorytet niż operator `*` (bardziej „przyciąga” swoje argumenty).

8.1.5 Przykłady

Na listingu 8.6 przedstawiono przykładowy program, w którym do początkowo pustej listy wstawia się dopisując na początek kolejno liczby 8, 2 i 7.

Listing 8.6: Szereg wstawień na początek

```
// main1.c

#include <stdio.h>
#include "listy.h"

int main(void)
{
        Lista wsk = NULL;
        wsk = dopisz_na_poczatek(wsk, 8);
        wsk = dopisz_na_poczatek(wsk, 2);
        wsk = dopisz_na_poczatek(wsk, 7);
        printf("zawartość listy = ");
        drukuj(wsk);
        usun(&wsk);
        return 0;
}
```

Gdy wydrukuje się elementy tak utworzonej listy są one w następującej kolejności: `7 2 8`.

Z kolei na listingu 8.7 przedstawiono przykładowy program, w którym do początkowo pustej listy wstawia się dopisując na koniec kolejno liczby 8, 2 i 7.

Listing 8.7: Szereg wstawień na koniec

```
// main2.c

#include <stdio.h>
#include "listy.h"

int main(void)
{
        Lista wsk = NULL;
        dopisz_na_koniec(&wsk, 8);
        dopisz_na_koniec(&wsk, 2);
        dopisz_na_koniec(&wsk, 7);
        printf("zawartość listy = ");
        drukuj(wsk);
        usun(&wsk);
        return 0;
}
```

Gdy wydrukuje się elementy tak utworzonej listy są one w następującej kolejności: `8 2 7`.

8.2 Uporządkowane drzewa binarne

Uporządkowane drzewa binarne można zrealizować za pomocą struktury przedstawionej na listingu 8.8.

Listing 8.8: Deklaracje drzew

```
// drzewa.h

struct Węzeł {
        int wartość;
        struct Węzeł* lewe;
        struct Węzeł* prawe;
};

typedef struct Węzeł* Drzewo;

void drukuj_prefiksowo(Drzewo wsk);
void drukuj_infiksowo(Drzewo wsk);
void drukuj_postfiksowo(Drzewo wsk);
void wstaw_uporzadkowane(Drzewo* wsk, int i);
void usun(Drzewo* wsk);
```

Struktura ta ma trzy pola:

`wartość` pole przechowuje wartość znajdującą się w korzeniu danego drzewa,

`lewe` pole przechowuje wskazanie na lewe poddrzewo,

`prawe` pole przechowuje wskazanie na prawe poddrzewo.

Zdefiniowano w nim również typ `Drzewo` jako wskazanie na strukturę `Węzeł`.

Załóżmy, że zmienna `wsk` typu `Drzewo` wskazuje na uporządkowane drzewo przechowujące liczby $1, 2, 3, 4$. Na rysunku 8.4a przedstawiono przykładowe takie drzewo. Nie jest to jedyne możliwe takie drzewo. Na rysunku 8.4 przedstawiono inne możliwe uporządkowane przechowujące te same cztery liczby $1, 2, 3, 4$.

8.2.1 Wstawianie

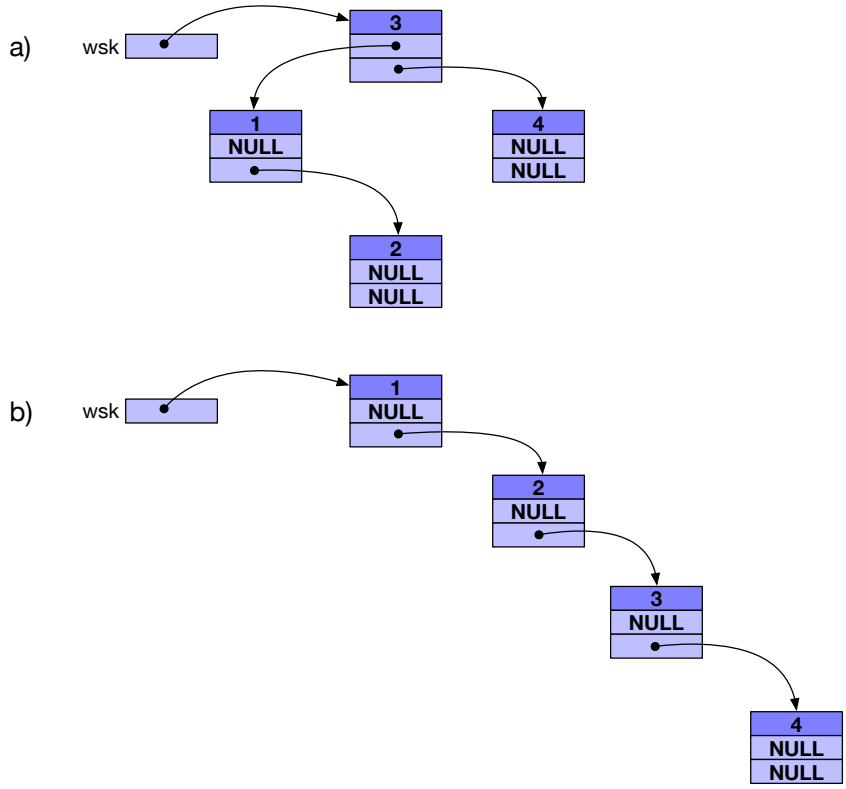
Na listingu 8.9 przedstawiono funkcję, która wstawia kolejną wartość do drzewa uporządkowanego zachowując jego uporządkowanie. Jeśli wstawiana wartość występowała już w drzewie, to nie zostanie wstawiona (każda wartość będzie występować w drzewie co najwyżej jeden raz).

Listing 8.9: Wstawianie do drzewa uporządkowanego

```
// wstaw_uporzadkowane.c

#include <stdlib.h>
#include "drzewa.h"

void wstaw_uporzadkowane(Drzewo* wsk, int i)
{
  if(*wsk == NULL)
    {
      *wsk = malloc(sizeof(struct Węzeł));
      (*wsk)->wartość = i;
      (*wsk)->lewe = NULL;
      (*wsk)->prawe = NULL;
    }
```



Rysunek 8.4: Przykłady drzew uporządkowanych.

```
        else
          if(i < (*wsk)->wartość)
            wstaw_uporzadkowane(&((*wsk)->lewe), i);
          else
            if(i > (*wsk)->wartość)
              wstaw_uporzadkowane(&((*wsk)->prawe), i);
}
```

Funkcja jest zapisana w postaci rekurencji. Gdy wstawiana wartość jest mniejsza od wartości znajdującej się w korzeniu, to zostaje ona wstawiona do lewego poddrzewa, natomiast gdy jest większa to do prawego. Zauważ, że gdy jest równa wartości w korzeniu, to funkcja kończy pracę nie wstawiając tej wartości po raz drugi.

8.2.2 Drukowanie drzewa

Aby wydrukować wartości przechowywane w drzewie należy przejść po jego wszystkich węzłach.

Możliwe są następujące trzy porządki obchodzenia węzłów drzewa:

prefiskowo najpierw przetwarza się korzeń a potem poddrzewo lewe a następnie prawe,

infiksowo najpierw przetwarza się poddrzewo lewe, potem korzeń a na końcu poddrzewo prawe,

postfiksowo najpierw przetwarza się poddrzewo lewem potem prawe a na koniec korzeń.

Dla przykładu, kolejność przetwarzania węzłów drzewa z rysunku 8.4a:

prefiksowo	3 1 2 4
infiksowo	1 2 3 4
postfiksowo	2 1 4 3

Zauważ, że drukując infiksowo wartości drzewa uporządkowanego, to będą one wydrukowane w kolejności rosnącej.

Zauważ również, że kolejność postfiksowa nie jest po prostu odwrotna do kolejności prefiksowej.

Na listingach 8.10-8.12 przedstawiono funkcje drukujące drzewo w trzech porządkach.

Listing 8.10: Drukowanie drzewa (prefiksowo)

```
// drukuj_prefiksowo.c

#include <stdio.h>
#include "drzewa.h"

void drukuj_prefiksowo(Drzewo wsk)
{
  if(wsk != NULL)
    {
      printf("%d ", wsk->wartość);
      drukuj_prefiksowo(wsk->lewe);
      drukuj_prefiksowo(wsk->prawe);
```

```
    }
}
```

Listing 8.11: Drukowanie drzewa (infiksowo)

```
// drukuj_infiksowo.c

#include <stdio.h>
#include "drzewa.h"

void drukuj_infiksowo(Drzewo wsk)
{
  if(wsk != NULL)
    {
      drukuj_infiksowo(wsk->lewe);
      printf("%d ", wsk->wartość);
      drukuj_infiksowo(wsk->prawe);
    }
}
```

Listing 8.12: Drukowanie drzewa (postfiksowo)

```
// drukuj_postiksowo.c

#include <stdio.h>
#include "drzewa.h"

void drukuj_postfiksowo(Drzewo wsk)
{
  if(wsk != NULL)
    {
      drukuj_postfiksowo(wsk->lewe);
      drukuj_postfiksowo(wsk->prawe);
      printf("%d ", wsk->wartość);
    }
}
```


8.2.3 Wyszukiwanie w drzewie uporządkowanym

Aby wyszukać wartość w drzewie uporządkowanym wystarczy porównać ją z wartością w korzeniu i jeśli jest różna, to dalsze poszukiwania przenieść do jednego z poddrzew w zależności od tego czy jest ono mniejsza od wartości w korzeniu (do lewego) albo większa (do prawego).

Na poniższym listingu przedstawiono funkcję, która dla danego drzewa wskazywanego pierwszym parametrem sprawdza czy wartość zadana drugim występuje w drzewie:

```
#include <stdbool.h>
#include "drzewa.h"

bool znajdz(Drzewo wsk, int x)
{
  if(wsk == NULL)
    return false;
```

```
  if(wsk->wartość == x)
    return true;
  else
    if(x < wsk->wartość)
      return znajdz(wsk->lewe, x);
    else
      return znajdz(wsk->prawe, x);
}
```

Liczba kroków jakie wykonuje funkcja `znajdz()` jest ograniczona przez wysokość drzewa. Im drzewo jest niższe, tym szybciej jesteśmy w stanie rozstrzygnąć czy szukana w nim wartość występuje. Z tego powodu ważne jest by duże zbiory wartości przechowywać w drzewach zrównoważonych. Dzięki temu liczba kroków potrzebnych do wyszukania wartości jest funkcją logarytmiczną liczby wartości przechowywanych w drzewie, a ta rośnie dużo wolniej niż liniowa (będzie o tym w rozdziale 10).

8.2.4 Usuwanie drzewa

Zanim usunie się węzeł przechowujący korzeń drzewa, należy usunąć wcześniej jego oba poddrzewa. Usunięcie korzenia bez usunięcia jego poddrzewa powodowałoby bezpowrotną utratę pamięci (tzw. *wyciek pamięci*). To dlatego, że gdy utraci się wskazanie na strukturę nie można już odzyskać pamięci zajmowanej przez nią.

Na listingu 8.13 przedstawiono funkcję usuwającą całe drzewo z pamięci. Jest ona zapisana w postaci rekurencji. Parametr `wsk` wskazuje na miejsce pamięci gdzie jest wskaźnik prowadzący do usuwanego drzewa. Dzięki temu podstawiając stałą `NULL` za `*wsk` nie dojdzie do sytuacji, w której jakiś wskaźnik prowadzi do nieistniejącej już struktury (tzw. *wiszący wskaźnik*).

Listing 8.13: Usuwanie drzewa

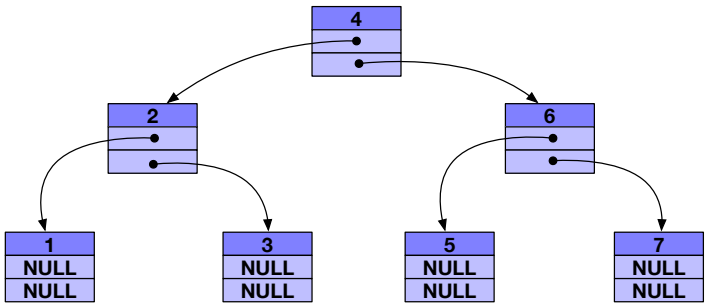
```
// usun.c

#include <stdlib.h>
#include "drzewa.h"

void usun(Drzewo* wsk)
{
  if(*wsk != NULL)
    {
      usun(&((*wsk)->lewe));
      usun(&((*wsk)->prawe));
      free(*wsk);
      *wsk = NULL;
    }
}
```

Zwróć uwagę na konieczność pobrania adresu pól w strukturze wskazywanej przez wskaźnik `*wsk`.

Na ćwiczeniach postaramy się napisać funkcję usuwającą drzewo za pomocą iteracji a nie przez rekurencję.



Rysunek 8.5: Zrównoważone drzewo o siedmiu węzłach.

8.2.5 Przykład

Na listingu 8.14 przedstawiono przykładowy program korzystający z omówionych funkcji.

Listing 8.14: Porównanie porządków drukowania


```c
// main.c

#include <stdio.h>
#include "drzewa.h"

int main(void)
{
  Drzewo wsk = NULL;
  wstaw_uporzadkowane(&wsk, 4);
  wstaw_uporzadkowane(&wsk, 2);
  wstaw_uporzadkowane(&wsk, 3);
  wstaw_uporzadkowane(&wsk, 6);
  wstaw_uporzadkowane(&wsk, 1);
  wstaw_uporzadkowane(&wsk, 7);
  wstaw_uporzadkowane(&wsk, 5);
  printf(" pre = ");
  drukuj_prefiksowo(wsk);
  printf("\n  in = ");
  drukuj_infiksowo(wsk);
  printf("\npost = ");
  drukuj_postfiksowo(wsk);
  printf("\n");
  usun(&wsk);
  return 0;
}
```

Na początek buduje on uporządkowane drzewo binarne. Zwróć uwagę, że wstawiając wartości w kolejności `4 2 3 6 1 7 5` otrzyma się idealnie zrównoważone drzewo o wysokości 3 (patrz rysunek 8.5).

Wynik uruchomienia programu:

```
$ ./main
 pre = 4 2 1 3 6 5 7
```

```
  in = 1 2 3 4 5 6 7
post = 1 3 2 5 7 6 4
```