Marek Gągolewski
INSTYTUT BADAŃ SYSTEMOWYCH PAN
WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH POLITECHNIKI WARSZAWSKIEJ

Algorytmy
i podstawy programowania

7. Dynamiczne struktury danych

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

7.1. Wprowadzenie

Wróćmy do zagadnienia implementacji podstawowych abstrakcyjnych typów danych wprowadzonych w rozdz. 5, tj.:

1. stosu (rozdz. 7.2),
2. kolejki (rozdz. 7.3),
3. kolejki priorytetowej (rozdz. 7.4),
4. słownika (rozdz. 7.5).

Przypomnijmy, że definicje wszystkich abstrakcyjnych typów danych *abstrahują od sposobu ich implementacji*. Do tej pory zastanawialiśmy się, w jaki sposób zaimplementować je z użyciem tablicy.

Informacja

Tak jak ostatnio, bez straty ogólności umawiamy się, że każdy prezentowany abstrakcyjny typ danych będzie służył do przechowywania pojedynczych wartości całkowitych, tj. danych typu `int`.

7.2. Stos

Stos (ang. *stack*) jest abstrakcyjnym typem danych typu LIFO (ang. *last-in-first-out*), który udostępnia przechowywane weń elementy w kolejności odwrotnej niż ta, w której były one zamieszczane.

Stos udostępnia trzy operacje:

- *push* (umieść) — wstawia element na początek stosu,
- *pop* (zdejmij) — usuwa i zwraca element z początku stosu,
- *empty* (czy pusty) — sprawdza, czy na stosie nie znajdują się żadne elementy.

Chciałoby się rzec — tylko tyle i aż tyle. W jaki sposób zaimplementować taki ATD? Dysponując wiedzą, którą posiadamy z poprzednich wykładów, najprawdopodobniej spróbowalibyśmy stworzyć stos oparty na zwykłej tablicy.

7.2.1. Implementacja I: tablica

Do stworzenia najprostszej implementacji stosu potrzebować będziemy trzech zmiennych (zadeklarowanych np. w funkcji `main()`[1]):

```
int  n = ...;          // maksymalna liczba elementów na stosie
int* s = new int[n]; // tu przechowujemy elementy
int  t = -1;           // indeks aktualnego ostatniego elementu
```

Zacznijmy od prostego przykładu użycia stosu.

[1] Lub umieszczonych w jednej strukturze `struct Stos{ ... };`

```
int main()
{
    int  n = 100;
    int* s = new int[n];
    int  t = -1;

    push(7, s, t, n);
    push(5, s, t, n);
    push(3, s, t, n);

    while (!empty(s, t, n))
       cout << pop(s, t, n);
    // wypisze na ekran 357

    return 0;
}
```

Rzecz jasna, najprostszą do zrealizowania operacją jest sprawdzenie, czy stos jest pusty:

```
bool empty(int* s, int t, int n)
{
    return (t < 0); // true, jeśli pusty
}
```

Operacja zdejmowania elementu ze stosu może wyglądać tak:

```
int pop(int* s, int& t, int n)
{
    assert(t >= 0); // zakładamy, że mamy co zdejmować
    return s[t--];  // zwróć s[t], a potem zmniejsz t o 1
}
```

Zauważmy, że zmienna `t` została przekazana przez referencję. Chcemy, by jej zmiana była „widoczne na zewnątrz”.

Na koniec operacja dodawania elementu do stosu:

```
void push(int x, int* s, int& t, int n)
{
    assert(t < n-1); // zakładamy, że mamy gdzie umieścić element
    s[++t] = x;      // najpierw zwiększ t o 1, a potem s[t] = x;
}
```

Widzimy, że wszystkie operacje potrzebują zawsze stałej liczby dostępów do tablicy, tzn. są rzędu $O(1)$. Jest to *rozwiązanie optymalne* (bardziej efektywne z obliczeniowego punktu widzenia istnieć nie może).

Napotykamy w tym miejscu jednak istotny problem — *w definicji naszego ATD nie pojawia się wcale założenie, że liczba przechowywanych elementów ma być z góry ograniczona*. Z tego powodu powyższa implementacja stosu jest tylko częściowo zgodna z definicją (formalista rzekłby, że to wcale nie jest implementacja stosu). Rzecz jasna, w przypadku niektórych programów takie ograniczenie wcale nie musi być niepoprawne. Co się stanie, jeśli naprawdę nie możemy z góry ustalić, ile elementów powinien móc przechowywać stos?

7.2.2. Implementacja II: tablica rozszerzalna

Spróbujmy zatem operacji dodawania elementów do stosu na powiększanie (w razie potrzeby) tablicy `s`. Będziemy więc musieli ją przekazywać przez referencję, gdyż może ona

(a właściwie adres jej pierwszego elementu) ulec zmianie. Co więcej, także `n` może wówczas się zmienić.

```
void push(int x, int*& s, int& t, int& n)
{
   if (t == n-1)
   {
      int* starys = s;  // zapamiętaj, co było do tej pory
      s = new int[n+1]; // nowa tablica - o jedno miejsce więcej (
         na przykład)

      for (int i=0; i<n; ++i)
         s[i] = starys[i];  // przepisz cały ,,stary stos''

      ++n;                  // rozmiar jest już większy

      delete [] starys;     // stare już niepotrzebne
   }

   s[++t] = x;
}
```

Niestety, *pesymistyczna złożoność obliczeniowa operacji push wynosi teraz* $O(n)$ (dlaczego?). Co więcej (powyższa implementacja dzieli tę własność z poprzednią), jeśli stos jest jedynie częściowo wypełniony, marnotrawimy wiele cennych komórek pamięci tylko po to, by były one w gotowości „na zaś". Jeśli zaprogramujemy dodatkowo operację *pop* tak, by zmniejszała tablicę `s`, to i ta operacja stanie się bardziej kosztowna obliczeniowo. Taki stan rzeczy nie może nam się podobać, spróbujemy więc poszukać lepszego rozwiązania.

7.2.3. Implementacja III: lista jednokierunkowa

Przedstawimy teraz implementację stosu, która wykorzystuje tzw. *listę jednokierunkową*. Istotną zaletą takie rozwiązania jest to, że wszystkie 3 omawiane operacje są realizowane w stałym czasie ($O(1)$) oraz że liczba przechowywanych elementów jest dowolna.

Lista jednokierunkowa to dynamiczna struktura danych, która składa się z *węzłów* reprezentowanych przez typ złożony o następującej definicji:

```
struct wezel
{
   int elem;     // element przechowywany w węźle
   wezel* nast; // wskaźnik na następny element
};
```

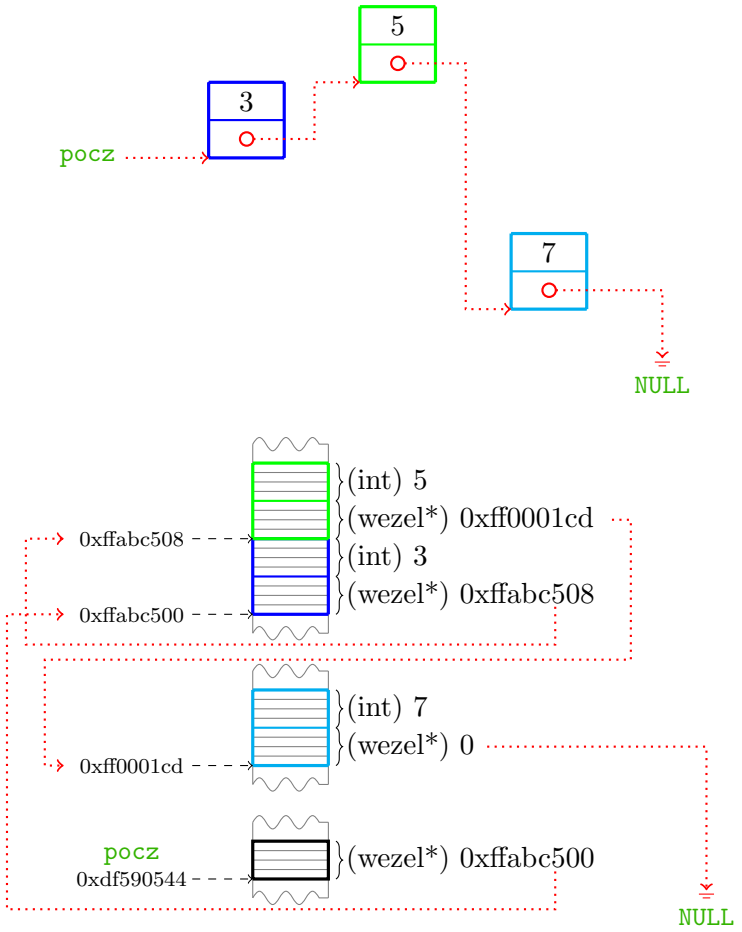
Zauważmy, że taka definicja ma w pewnym sensie charakter rekurencyjny. W definicji węzła pojawia się odwołanie do innego węzła (za pomocą wskaźnika). Węzły w liście są ze sobą połączone, jeden za drugim. Do stworzenia listy wystarczy zatem wskaźnik na jej pierwszy element (tzw. *głowa*):

```
wezel* pocz;
```

oraz informacja, po którym węźle lista się kończy. Do tego celu stosuje się najczęściej wartownika — adres `NULL` (0).

Schemat przykładowej listy jednokierunkowej przechowującej elementy 3, 5 oraz 7 przedstawia rys. 7.1. Zwróćmy szczególną uwagę na to, w jaki sposób węzły *mogą być* rozlokowane w pamięci.

Nim zaczniemy się zastanawiać, w jaki sposób zrealizować stos za pomocą listy, zacznijmy od stworzenia pliku nagłówkowego (np. `stos.h`).



Rys. 7.1. Przykładowa lista jednokierunkowa przechowująca elementy 3,5,7. Schemat graficzny (powyżej) i przykładowa organizacja pamięci (poniżej).

```
#pragma once

struct wezel
{
    int elem;     // element przechowywany w węźle
    wezel* nast; // wskaźnik na następny element
};

void push(int i, wezel*& pocz);
int  pop(wezel*& pocz);
bool empty(wezel* pocz);
```

Zakładamy tutaj, że operacje *push* i *pop* mogą spowodować zmianę pierwszego elementu listy.

Oto przykładowy plik `main.cpp`, zawierający funkcję `main()` analogiczną do tej, którą zaprezentowaliśmy dla implementacji tablicowej stosu.

```
#include ...

int main()
{
    wezel* pocz = NULL; // pusta lista (na początek)

    push(7, pocz);
```

```
8     push(5, pocz);
9     push(3, pocz);
10
11    while (!empty(pocz))
12       cout << pop(pocz);
13
14    // 357
15
16    return 0;
17 }
```

Znowu najprostszą funkcją będzie operacja sprawdzająca, czy stos jest pusty. Wystarczy sprawdzić, czy węzeł początkowy wskazuje bezpośrednio na wartownika.

```
bool empty(wezel* pocz)
{
   return (pocz == NULL); // true, jeśli lista pusta
}
```

Operacje *push* i *pop* można zrealizować jako, odpowiednio, *wstawianie elementu na początek oraz usuwanie elementu z początku listy*. Łatwo się domyślić, że będą one potrzebowały stałej liczby dostępów do węzłów listy, tzn. będą rzędu $O(1)$.

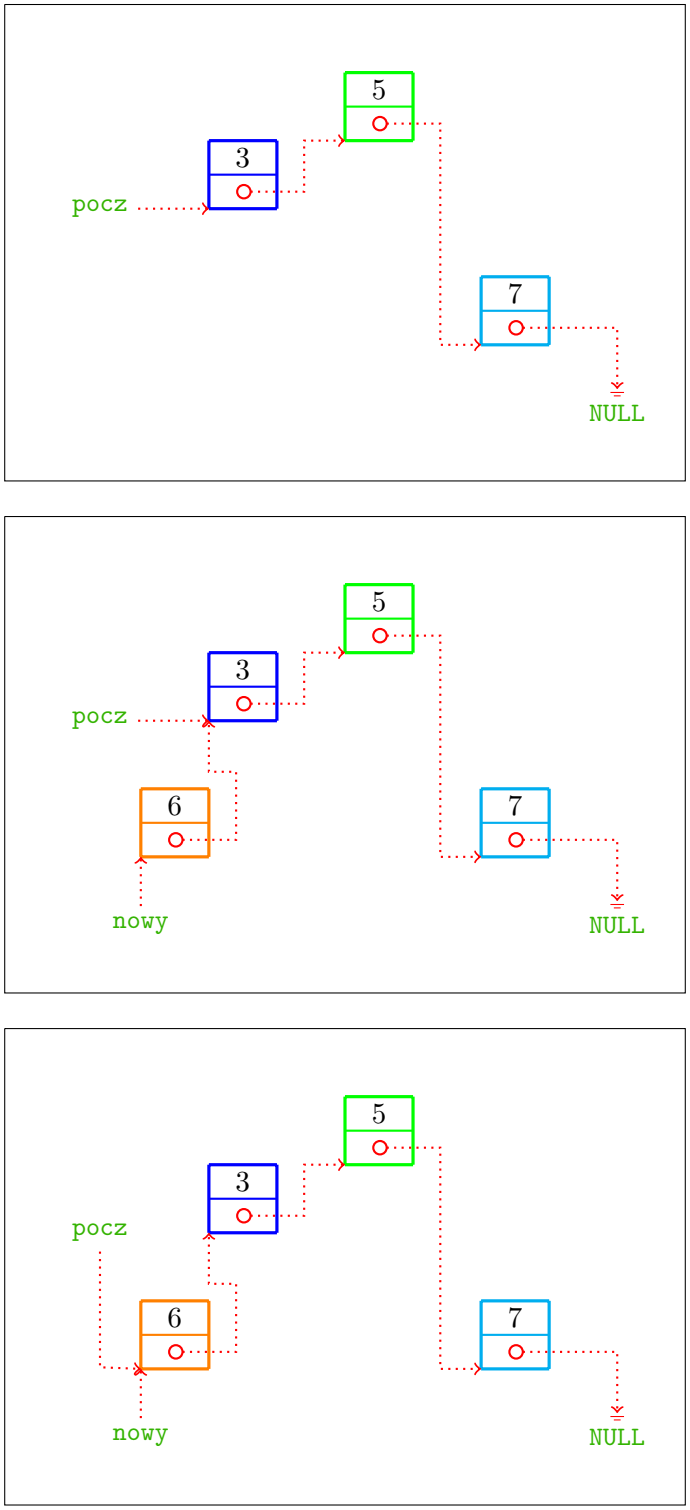
Najpierw zajmijmy się wstawianiem elementu na początek listy. Rys. 7.2 ilustruje kolejne kroki potrzebne do utworzenia listy (6,3,5,7) z listy (3,5,7). Zwróćmy uwagę, że po dokonaniu tej operacji zmienia się głowa listy.

Oto przykładowa implementacja operacji *push* stosująca wstawianie elementu na początek listy jednokierunkowej. Zauważmy raz jeszcze, że pierwszym parametrem jest referencja na zmienną wskaźnikową. Przekazujemy tutaj głowę listy, która będzie zmieniona przez tę funkcję.

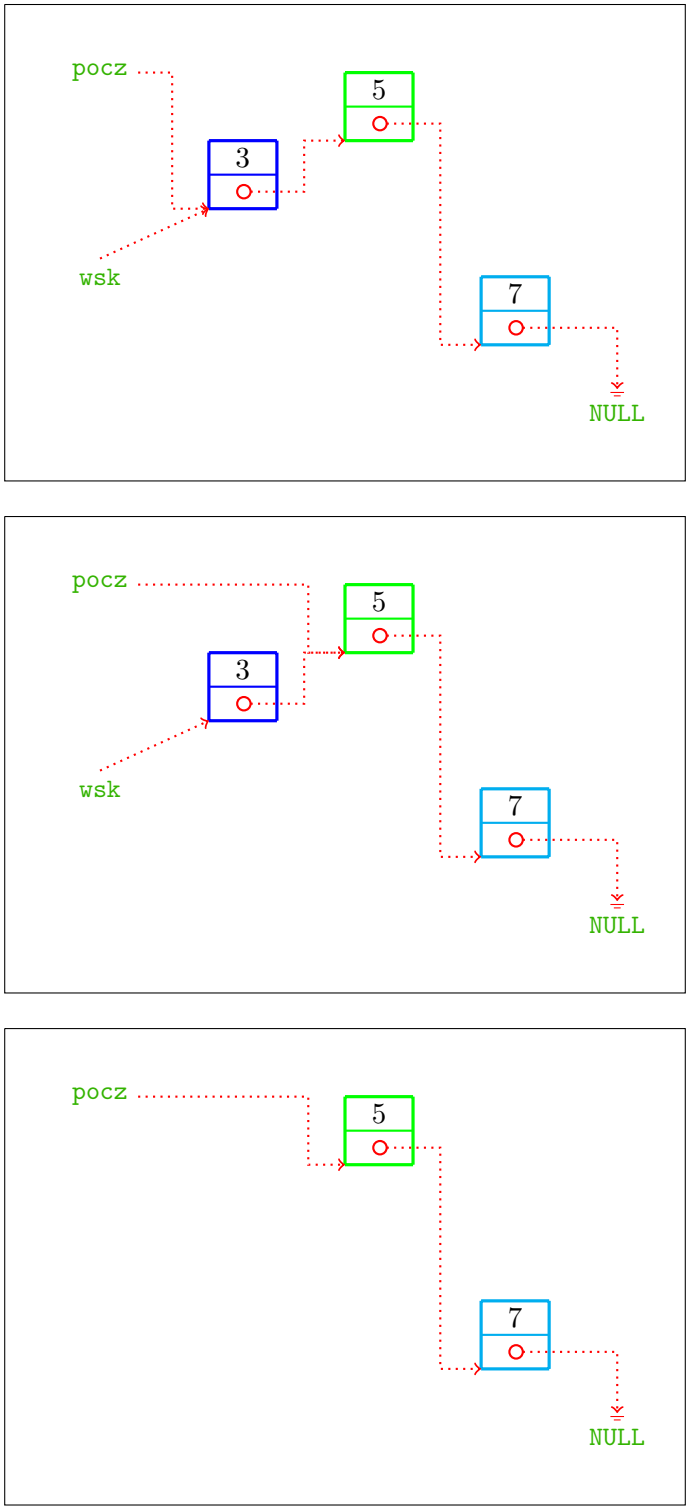
```
void push(int i, wezel*& pocz)
{
   wezel* nowy = new wezel;
   nowy->elem = i;

   nowy->nast = pocz; /* wskazuj na to, na co wskazuje pocz, może
      być NULL */

   pocz = nowy; /* teraz lista zaczyna się od nowego węzła */
}
```

Rys. 7.3 przedstawia możliwy sposób usuwania elementów z początku listy jednokierunkowej. Implementację operacji *pop* przedstawia poniższy kod. Zauważmy, że po jej dokonaniu możliwe jest osiągnięcie stanu `glowa == NULL`, co oznacza, że usunięty element był jedynym.

```
int pop(wezel*& pocz)
{
   assert(pocz != NULL); // na wszelki wypadek...

   int i = pocz->elem;    // zapamiętaj przechowywany element

   wezel* wsk = pocz;     // zapamiętaj "stary" pocz
   pocz = pocz->nast;     // zmień pocz
```



Rys. 7.2. Wstawianie elementu 6 na początek listy jednokierunkowej.



Rys. 7.3. Usunięcie elementu z początku listy jednokierunkowej.

```
    delete wsk;             // skasuj "stary" pocz

    return i;
}
```

Jak widzimy, powyższe operacje wykonywane na liście jednokierunkowej wcale nie są (wbrew obiegowej opinii) skomplikowane. Ważne jest, byśmy już teraz dobrze zrozumieli zasadę ich działania.

W następnym podrozdziale omówimy kolejny abstrakcyjny typ danych — kolejkę. Okaże się, że jej efektywną implementację można także oprzeć na (odpowiednio zmodyfikowanej) liście jednokierunkowej.

7.3. Kolejka

Kolejka (ang. *queue*) to ATD typu FIFO (ang. *first-in-first-out*). Podstawową własnością kolejki jest to, że pobieramy z niej elementy w takiej samej kolejności (a nie w odwrotnej jak w przypadku stosu), w jakiej były one umieszczane.

Kolejka udostępnia następujące trzy operacje:

- *enqueue* (umieść) — wstawia element na koniec kolejki,
- *dequeue* (wyjmij) — usuwa i zwraca element z początku kolejki,
- *empty* (czy pusty).

Zadanie

Jak łatwo się domyślić, implementacje kolejki oparte na tablicy będą cechować się podobnymi wadami, jakimi cechowały się analogiczne rozwiązania z rozdz. 7.2.1 i 7.2.2. Ich wykonanie pozostawmy wobec tego zacnemu Czytelnikowi. Będzie to dlań dobrym ćwiczeniem przed kolokwium.

7.3.1. Implementacja I: lista jednokierunkowa

Zastanówmy się, w jaki sposób można zrealizować operacje kolejki z użyciem listy jednokierunkowej. Mamy tutaj dwie możliwości:

1. *enqueue1* — wstaw element na początek listy, *dequeue1* — usuń z końca, bądź
2. *enqueue2* — wstaw element na koniec listy, *dequeue2* — usuń z początku.

Niestety, choć (co już wiemy) *enqueue1* / *dequeue2* wymagają $O(1)$ operacji, to pozostałe działania na liście cechują się pesymistyczną złożonością obliczeniową rzędu $O(n)$. Mimo to spróbujmy je zaimplementować.

Rozważmy najpierw, w jaki sposób wstawić element na koniec listy (działanie potrzebne do zrealizowania operacji *enqueue2*). Rys. 7.4 i 7.5 ilustrują, jak utworzyć listę $(3, 5, 7, 1)$, mając daną listę $(3, 5, 7)$. Zauważmy, że pierwszym krokiem, jaki należy wykonać, jest przejście na koniec listy. Tutaj korzystamy z dwóch dodatkowych wskaźników.

Poniższa funkcja implementuje omawianą operację. Zauważmy, że drugim parametrem jest referencja do wskaźnika (a dokładniej — do głowy listy).

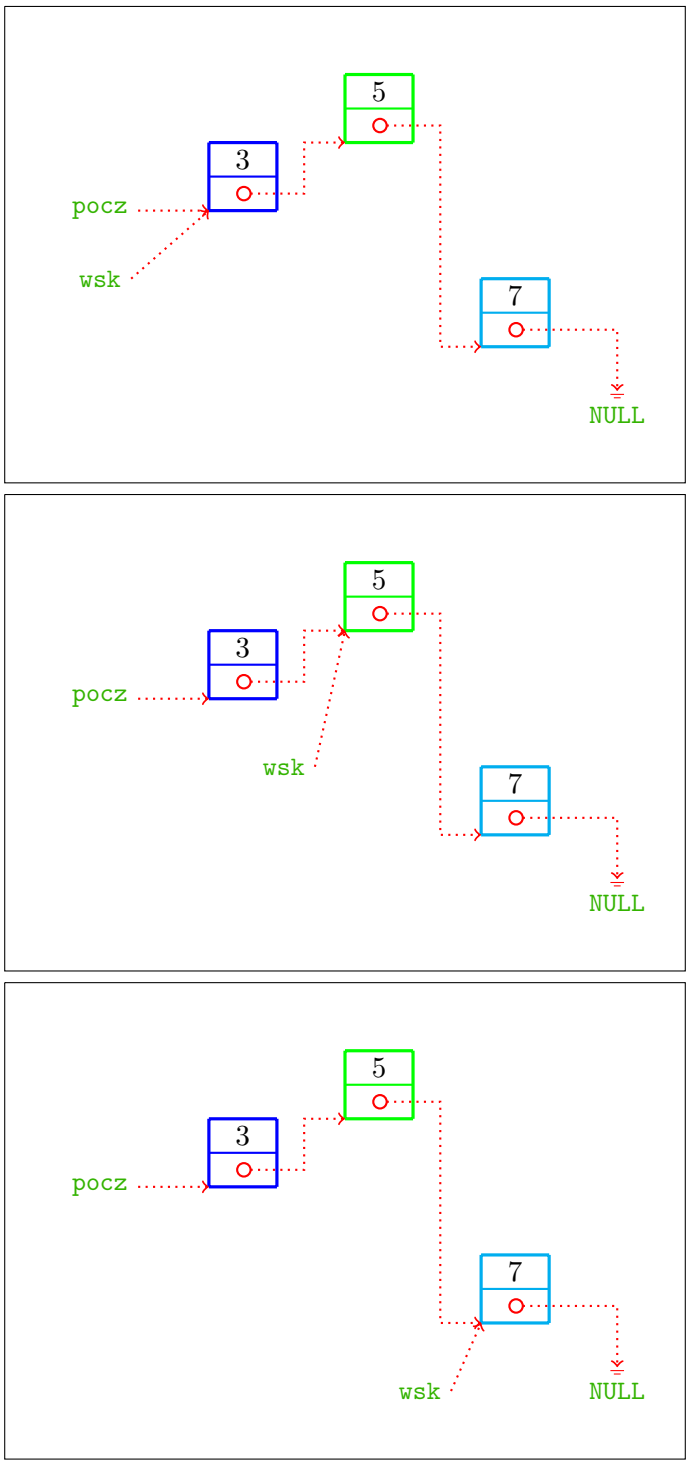
```
void enqueue2(int i, wezel*& pocz)
{
    wezel* nowy = new wezel;
    nowy->elem = i;
    nowy->nast = NULL; // to będzie ostatni element

    if (pocz == NULL) // czy lista pusta?
       pocz = nowy;
    else {
       wezel* wsk = pocz;
       while (wsk->nast != NULL)
          wsk = wsk->nast; // przejdź na ostatni element
       wsk->nast = nowy;   // nowy ostatni element
    }
}
```

A oto równoważna wersja rekurencyjna.

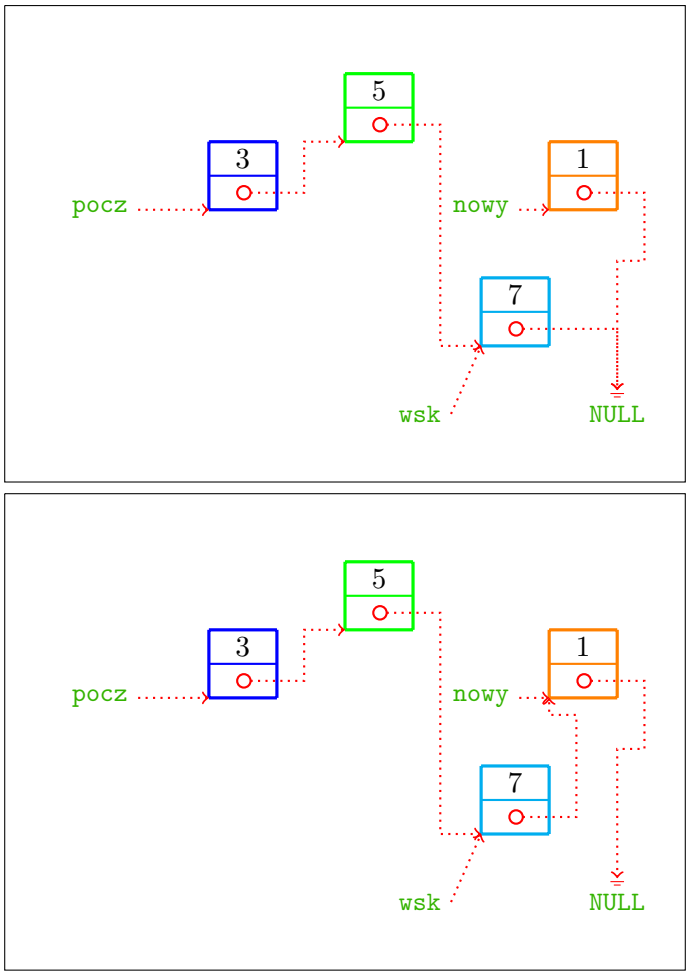
```
void enqueue2_rek(int i, wezel*& wsk) // referencja!
{
    if (wsk != NULL)                 // to nie jest koniec...
       enqueue2_rek(i, wsk->nast); // więc idź dalej
    else
    {                                // jesteś na końcu
       wsk = new wezel;
       wsk->elem = i;
       wsk->nast = NULL;
    }
}
```

Pora na operację *dequeue1*, realizowaną za pomocą usuwania elementu z końca listy. Rys. 7.6 ilustruje przykładowy przebieg takiego procesu. W wersji iteracyjnej tej funkcji musimy przejść na przedostatni element listy. Należy się jednak wcześniej upewnić, czy przypadkiem nie kasujemy jedynego elementu (oddzielny przypadek).

```
int dequeue1(wezel*& pocz)
{
    assert (pocz != NULL);

    int i;
    if (pocz->nast == NULL)    // tylko 1 element
    {
       i = pocz->elem;
       delete pocz;
       pocz = NULL;
    }
    else                       // więcej niz 1 element
    {
       wezel* wsk = pocz;
       while (wsk->nast->nast != NULL)
          wsk = wsk->nast;     // idź na przedost. el.
       i = wsk->nast->elem;
       delete wsk->nast;
       wsk->nast = NULL;
    }
    return i;
}
```



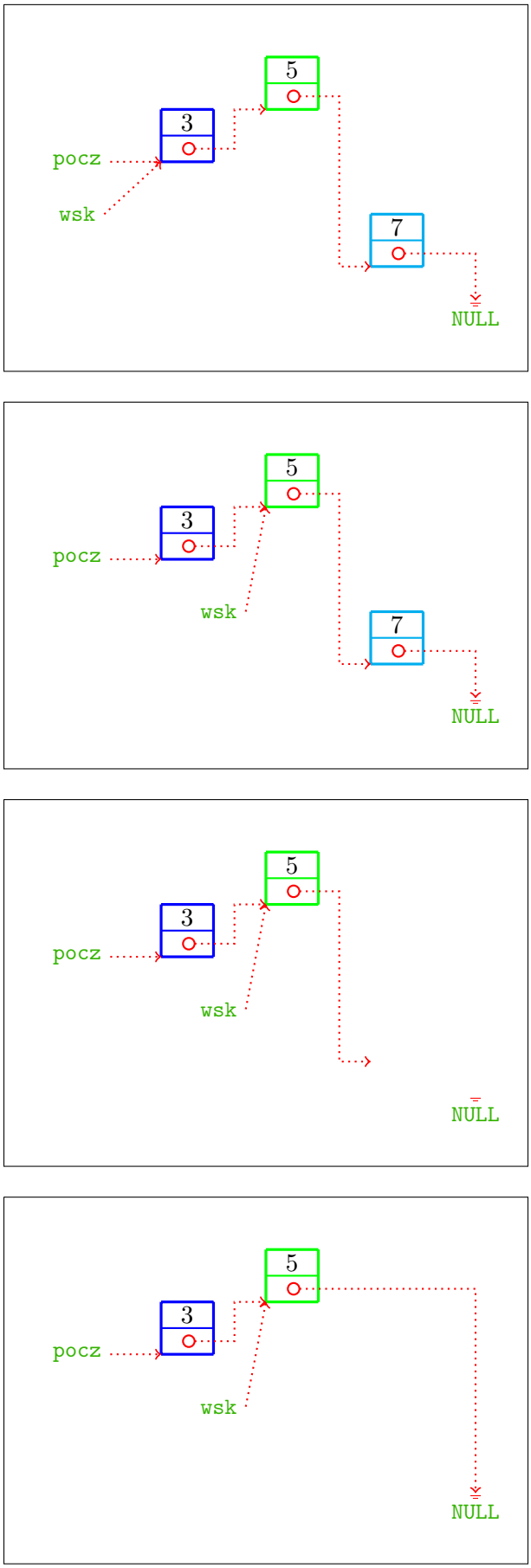
Rys. 7.4. Wstawianie elementu 1 na koniec listy jednokierunkowej cz. I.



Rys. 7.5. Wstawianie elementu 1 na koniec listy jednokierunkowej cz. II.

Omawianą procedurę również można sformułować znacznie prościej za pomocą rekurencji.

```
1  int dequeue1_rek(wezel*& wsk)
2  {
3      assert(wsk != NULL);
4
5      if (wsk->nast == NULL) // jesteśmy na końcu
6      {
7          int i = wsk->elem;
8          delete wsk;
9          wsk = NULL;
10         return i;
11     }
12     else // idziemy dalej
13         return dequeue1_rek(wsk->nast);
14 }
```



Rys. 7.6. Usunięcie elementu z końca listy jednokierunkowej.

7.3.2. Implementacja II: lista jednokierunkowa z „ogonem”

Okazuje się jednak, że operacje kolejki da się zaimplementować tak, by potrzebowały zawsze stałej liczby ($O(1)$) dostępów do elementów. W poprzednim rozwiązaniu krytyczną (tzn. najbardziej czasochłonną) czynnością było przejście na koniec listy. Gdybyśmy zapamiętali dodatkowo wskaźnik na ostatni element listy (ogon), wstawianie mogłoby się wykonywać bardzo szybko.

Zadanie

Ogon nie pozwala przyspieszyć operacji usuwania ostatniego elementu. Dlaczego?

Przyjrzyjmy się pełnej implementacji naszej kolejki. Deklaracje funkcji (w pliku `.h`) mogą wyglądać tak:

```
void enqueue(int i, wezel*& pocz, wezel*& ost);
int  dequeue(wezel*& pocz, wezel*& ost);
bool empty(wezel* pocz, wezel* ost);
```

W dwóch pierwszych przypadkach dopuszczamy zarówno możliwość zmiany głowy jak i ogona listy.

Przykładowa funkcja główna:

```
int main()
{
   wezel* pocz = NULL; // pusta lista (na początek)
   wezel* ost = NULL;  // wskaźnik na ostatni element — też NULL

   for (int i=0; i<10; ++i) // wstaw coś w celach testowych
      enqueue(i, pocz, ost);

   while (!empty(pocz, ost))
      cout << dequeue(pocz, ost);

   return 0;
}
```

Na koniec — definicje funkcji. Tak może wyglądać sprawdzenie, czy kolejka jest pusta:

```
bool empty(wezel* pocz, wezel* ost)
{
   return (pocz == NULL); // true, jeśli lista pusta
}
```

Umieszczenie elementu w kolejce:

```
void enqueue(int i, wezel*& pocz, wezel*& ost)
{
   // wstawianie na koniec listy

   wezel* nowy = new wezel; // stwórz nowy wezel
   nowy->elem = i;
   nowy->nast = NULL;

   if (pocz == NULL)        // gdy lista pusta...
   {
      pocz = ost = nowy;    // zmieniamy pocz i ost
   }
   else                     // gdy lista niepusta...
   {
      ost->nast = nowy;     // po ostatnim - nowy
      ost = nowy;           // teraz ostatni to nowy
   }

   // koszt operacji: O(1)
}
```

Wyjęcie elementu z kolejki:

```
int dequeue(wezel*& pocz, wezel*& ost)
{
   assert(pocz != NULL); // na wszelki wypadek...

   // usuwanie z początku listy

   int i = pocz->elem;   // zapamiętaj przechowywany element

   wezel* wsk = pocz;    // zapamiętaj "stary" pocz
   pocz = pocz->nast;    // zmień pocz
   delete wsk;           // skasuj "stary" pocz

   if (pocz == NULL) ost = NULL;

   return i;

   // koszt operacji: O(1)
}
```

Dzięki prostej modyfikacji listy (jednej dodatkowej informacji) udało nam się znacząco przyspieszyć działanie kolejki. W następnym podrozdziale wprowadzimy nowy ATD — bardziej zaawansowaną kolejkę, w której będą mogły występować obiekty w pewnym sensie „uprzywilejowane”.

7.4. Kolejka priorytetowa

Kolejka priorytetowa (ang. *priority queue*) to abstrakcyjny typ danych typu HPF (ang. *highest-priority-first*). Może ona służyć do przechowywania elementów, na których została określona pewna relacja porządku liniowego <=[2]. Elementy wydobywa się z niej

[2] W naszym przypadku, jako że umówiliśmy się, iż omawiane ATD przechowują dane typu `int`, najczęściej będzie to po prostu zwykły porządek $\leqslant$.

poczynając od tych, które mają największy priorytet (są maksymalne względem relacji <=). Jeśli w kolejce priorytetowej znajdują się elementy o takich samych priorytetach, obowiązuje dla nich kolejność FIFO, tak jak w przypadku zwykłej kolejki.

Omawiany ATD udostępnia trzy operacje:

– *insert* (włóż) — wstawia element na odpowiednie miejsce kolejki priorytetowej.

– *pull* (pobierz) — usuwa i zwraca element maksymalny,

– *empty* (czy pusty).

Zadanie

Podobnie jak wcześniej, implementacja tej struktury danych z użyciem tablicy wymaga $O(n)$ dostępów do obiektów w przypadku operacji *insert* oraz $O(1)$ dostępów przy wywołaniu *pull*. Pozostawmy jeszcze raz to pouczające zagadnienie jako wyzwanie dla zdolnego i pracowitego Czytelnika.

7.4.1. Implementacja I: lista jednokierunkowa

Spróbujmy zaimplementować kolejkę priorytetową używając listy jednokierunkowej, która przechowuje elementy posortowane względem relacji <= (w kolejności odwrotnej, tzn. największy element jest pierwszy). Operacja *pull* powinna więc po prostu zwracać i usuwać aktualną głowę listy (patrz wyżej, złożoność $O(1)$).

Oto wersja rekurencyjna operacji *insert*, która realizuje wstawianie elementu na odpowiednim miejscu listy (bez zaburzenia istniejącego porządku).

```
void insert(int i, wezel*& pocz)
{
   if (pocz == NULL || i > pocz->elem) // >, bo FIFO, gdy ==
   {     // wstawiamy tutaj
      wezel* starypocz = pocz; // może być NULL

      pocz = new wezel; // stwórz nowy wezel
      pocz->elem = i;
      pocz->nast = starypocz;
   }
   else // idziemy dalej
      insert(i, pocz->nast);
}
```

Niestety, powyższa operacja cechuje się liniową ($O(n)$) złożonością obliczeniową. Nie pomoże tu oczywiście uwzględnienie np. dodatkowego wskaźnika na ostatni element listy. Dostrzegamy wobec tego konieczność zastosowania w tym miejscu innej dynamicznej struktury danych.

7.4.2. Implementacja II: drzewo binarne

Drzewo binarne bądź binarne drzewo poszukiwań (ang. *binary search tree*, BST), to dynamiczna struktura danych, w której elementy są uporządkowane zgodnie z pewnym porządkiem liniowym <=. Dane przechowuje się tutaj w węzłach zdefiniowanych następująco:

```
struct wezel
{
    int elem;      // element przechowywany w węźle
    wezel* lewy;   // lewe podrzewo (lewy potomek)
    wezel* prawy; // prawe podrzewo (lewy potomek)
};
```

Widzimy więc, że każdy węzeł ma co najwyżej dwóch potomków (równoważnie: rodzic ma co najwyżej dwoje dzieci).

Drzewo binarne pozwala szybko (przeważnie[3], zob. dalej) wyszukiwać, wstawiać i usuwać elementy (średnio $O(\log_2 n)$[4], pesymistycznie $O(n)$).

Każde drzewo poszukiwań binarnych cechuje się następującymi własnościami (aksjomaty BST):

1. Drzewo ma strukturę *acykliczną*, tzn. wychodząc z dowolnego węzła za pomocą wskaźników nie da się do niego wrócić.
2. Niech v będzie dowolnym węzłem. Wówczas:
   - wszystkie elementy w lewym poddrzewie v są nie większe (`<=`) niż element przechowywany w v,
   - wszystkie elementy w prawym poddrzewie v są większe (`>`) niż element przechowywany w v.

Ciekawostka

Czasem rozpatruje się drzewa, w których zabronione jest przechowywanie duplikatów elementów.

Początek (pierwszy węzeł) drzewa określa element zwany *korzeniem* (ang. *root*), por. analogię do głowy listy. Korzeń nie ma, rzecz jasna, żadnych przodków. Co więcej, z korzenia można dojść do każdego innego węzła (warunek *spójności*).

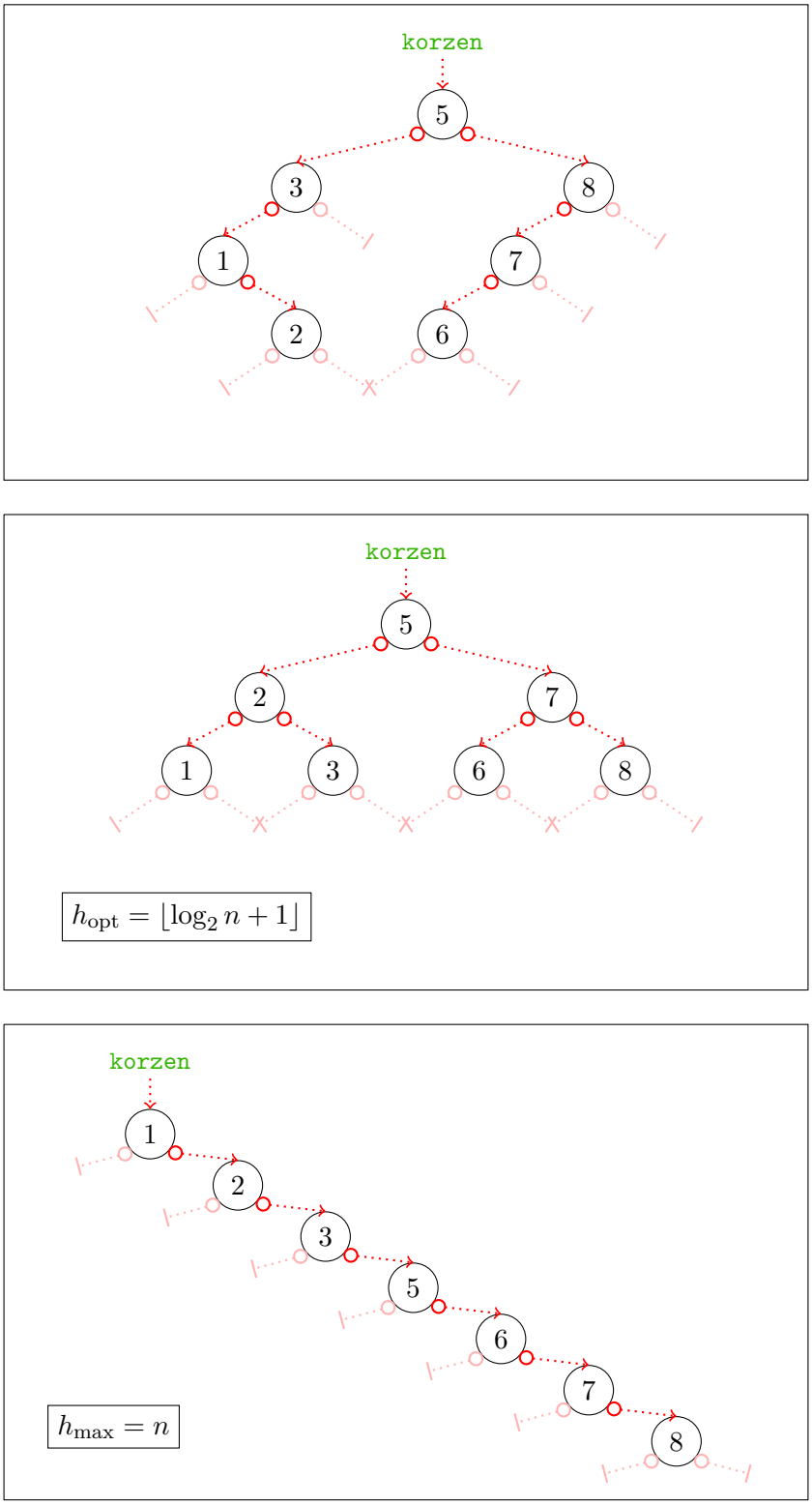
Węzły, które nie mają potomków (*węzły terminalne*, tzn. takie, których obydwaj potomkowie są ustaleni na `NULL`), zwane są inaczej *liśćmi* (ang. *leaves*).

Warto być świadomym faktu, iż drzewa *nie mają jednoznacznej postaci* — różne drzewa mogą przechowywać te same wartości. Rys. 7.7 przedstawia przykładowe BST, które zawierają elementy $1, 2, 3, 5, 6, 7, 8$. Drugie z nich jest *idealnie zrównoważone* — ma optymalną wysokość (liczbę poziomów). Liczba operacji koniecznych np. do znalezienia dowolnego elementu jest rzędu $O(\log_2 n)$. Z kolei trzecie drzewo jest *zdegenerowane do listy*. Wymaga $O(n)$ operacji w przypadku pesymistycznym.

Spróbujmy więc zaimplementować kolejkę priorytetową opartą na drzewie binarnym. Znowu najprostszą jest operacja *empty*:

[3] Studenci MiNI na przedmiocie Algorytmy i struktury danych (III sem.) poznają inne podobne struktury danych, min. drzewa AVL czy drzewa czerwono-czarne, które *gwarantują* wykonanie wymienionych operacji zawsze w czasie $O(\log_2 n)$. Dokonują one odpowiedniego równoważenia drzewa. Ciekawą alternatywą są także różne struktury danych typu kopiec.

[4] Logarytm jest funkcją, której wartości rosną dość wolno, np. $\log_2 1000 \simeq 10$, a $\log_2 10000 \simeq 13$ itd.



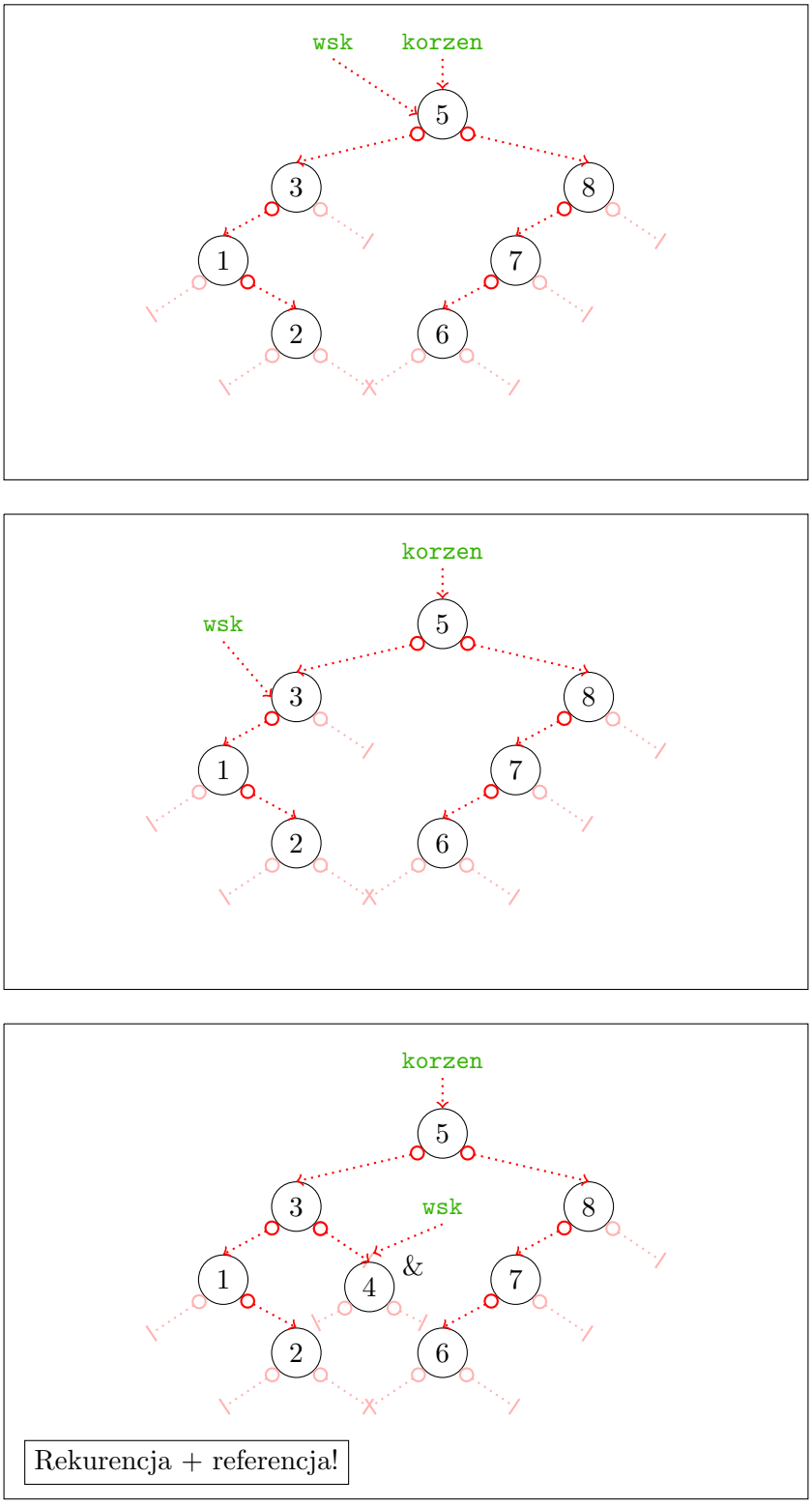
Rys. 7.7. Różne drzewa binarne przechowujące wartości $1, 2, 3, 5, 6, 7, 8$.

```
bool empty(wezel* pocz)
{
   return (pocz == NULL); // true, jeśli drzewo puste
}
```

Operacja *insert* powinna umieszczać nowy element jako liść „na swoim właściwym miejscu" (pamiętamy, że drzewo jest zawsze uporządkowane). Rys. 7.8 ilustruje wstawianie elementu o wartości 4, który powinien stać się lewym potomkiem węzła „5" ($4 \leqslant 5$). Nie będzie to jednak potomek bezpośredni, gdyż „honorowe" miejsce jest zajęte przez węzeł „3" (nie będziemy go „wydziedziczać"). „4" zostanie prawym ($4 > 3$) potomkiem „3" — bezpośrednim, bo „3" nie ma „większego" dziecka. Oto przykładowa implementacja rekurencyjna:

```
void insert(int i, wezel*& pocz)
{
   if (pocz == NULL)          // gdy poddrzewo puste...
   {
      pocz = new wezel;       // stwórz liść
      pocz->elem = i;
      pocz->lewy = NULL;
      pocz->prawy = NULL;
   }
   else if (i <= pocz->elem)
      insert(i, pocz->lewy);  // wstaw do lewego poddrzewa, będzie
         FIFO, gdy ==
   else
      insert(i, pocz->prawy); // wstaw do prawego poddrzewa
}
```

Pesymistyczny koszt wykonania tej operacji operacji jest rzędu $O(n)$ (dla drzewa zdegenerowanego), oczekujemy jednakże, że operacja wykonywać się będzie średnio w czasie $O(\log_2 n)$ (dla danych losowych).



Rys. 7.8. Wstawianie wartości 4 do drzewa binarnego.

Trzecia operacja, tzn. *pull*, wyszukiwać będzie największy element przechowywany w drzewie. Znajduje się on zawsze w najbardziej wysuniętym na prawo węźle (por. rys. 7.9). Przykładowa implementacja korzystająca z rekurencji:

```
int pull(wezel*& pocz)
{
   assert(pocz != NULL); // na wszelki wypadek...

   // usuwanie "skrajnie prawego" liścia
   if (pocz->prawy == NULL) // wtedy to, co w pocz >= od
       wszystkiego, co w całym poddrzewie
   {
      int i = pocz->elem;
      wezel* stary = pocz;
      pocz = pocz->lewy; // może być NULL
      delete stary;
      return i;
   }
   else
      return pull(pocz->prawy); // idź dalej
}
```

Koszt wykonania tej operacji jest taki sam jak w poprzednim przypadku.

7.5. Słownik

Słownik (ang. *dictionary*) to abstrakcyjny typ danych, który udostępnia trzy operacje:

- *search* (znajdź) — sprawdza, czy dany element znajduje się w słowniku,
- *insert* (dodaj) — dodaje element do słownika, o ile taki już się w nim nie znajduje,
- *remove* (usuń) — usuwa element ze słownika.

Dla celów pomocniczych rozważać będziemy także następujące działania na słowniku:

- *print* (wypisz),
- *removeAll* (usuń wszystko).

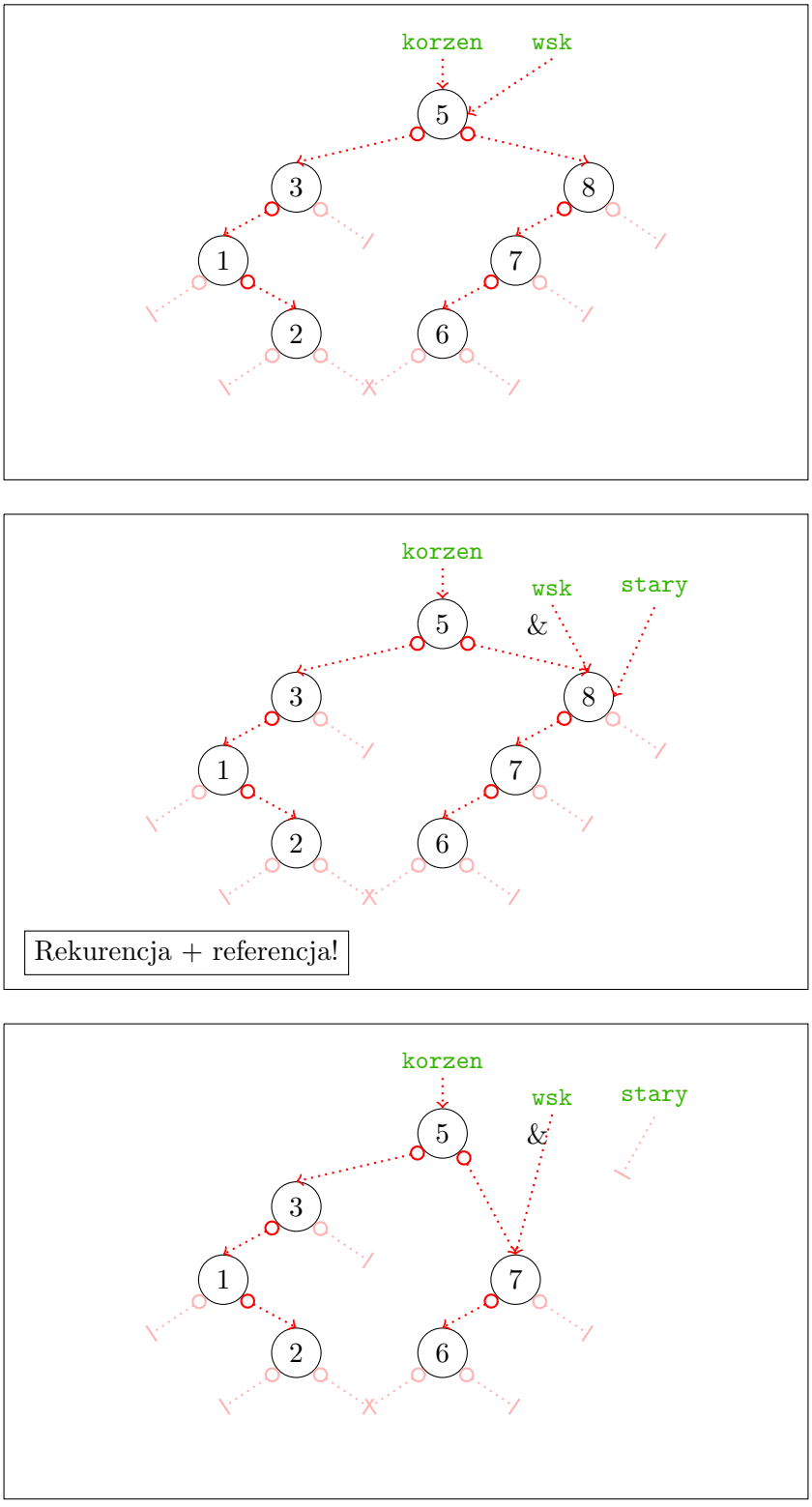
Informacja

Czasem zakłada się, że na przechowywanych elementach została określona relacja porządku liniowego <=. Wtedy możemy tworzyć bardziej efektywne implementacje słownika, np. z użyciem drzew binarnych (zob. dalej).

7.5.1. Implementacja I: tablica

Dość łatwo będzie nam zaimplementować słownik z użyciem zwykłej tablicy. Niestety, wszystkie operacje w swej najprostszej implementacji będą cechować się liniową ($O(n)$) pesymistyczną złożonością obliczeniową.

Jeśli jednak na elementach została określona relacja <= (co jest bardzo częstym przypadkiem), operację wyszukiwania można znacząco przyspieszyć przyjmując, że elementy są



Rys. 7.9. Usuwanie elementu największego w drzewie binarnym.

przechowywane zgodnie z powyższym porządkiem. Wymagane jest tu jedynie odpowiednie zaimplementowanie operacji *insert*).

W takiej sytuacji, w operacji *search* można skorzystać z algorytmu tzw. *wyszukiwania binarnego* (połówkowego), którego pesymistyczna złożoność wynosi zaledwie $O(\log n)$. Metoda ta bazuje na założeniu, że poszukiwany element `x`, jeśli oczywiście znajduje się w tablicy, musi być ulokowany na pozycji (indeksie) ze zbioru `{l,l+1,...,p}`. Na początku ustalamy oczywiście `l=0` i `p=n-1`. W każdej kolejnej iteracji znajdujemy „środek" owego obszaru poszukiwań, tzn. `m=(p+l)/2` oraz sprawdzamy, jak ma się `t[m]` do `x`. Jeśli okaże się, że jeszcze nie trafiliśmy na właściwe miejsce, to wtedy zawężamy (o co najmniej połowę) nasz obszar modyfikując odpowiednio wartość `l` bądź `p`.

```
bool search(int x, int* t, int n)
{
   if (n == 0) return false; // słownik pusty

   // wyszukiwanie binarne (połówkowe)
   int l = 0;     // początkowy obszar poszukiwań -
   int p = n-1;   //        - cała tablica
   while (l <= p)
   {
      int m = (p+l)/2; // "środek" obszaru poszukiwań

      if (x == t[m]) return true; // znaleziony - koniec
      else if (x > t[m]) l = m+1; // zawężamy obszar poszukiwań
      else               p = m-1; // jw.
   }

   return false; // nieznaleziony
}
```

Pozostałe operacje pozostawiamy drogiemu Czytelnikowi do samodzielnej implementacji.

7.5.2. Implementacja II: lista jednokierunkowa

Słownik także nietrudno jest zaimplementować na liście jednokierunkowej. Chociaż takie rozwiązanie jest mało efektywne, wszystkie operacje cechują się bowiem z konieczności złożonością $O(n)$, to jednak próba zmierzenia się z nim może być dla nas dobrym ćwiczeniem. Przedstawmy wobec tego gotowy kod wszystkich pięciu interesujących nas funkcji.

Zauważmy, że poza operacją *print* korzystamy tutaj w każdym przypadku z bardzo dla nas wygodnej rekurencji.

```
bool search(int x, wezel* w)
{
   if (w == NULL)
      return false; // koniec - nie ma
   else if (w->elem == x)
      return true;  // jest
   else
      return search(x, w->nast); // może jest dalej?
}
```

```
void insert(int x, wezel*& w)
{
   /* Gdyby w słowniku nie mogły występować duplikaty elementów,
      ta operacja cechowałaby się złożonością O(1)
      - można by było wstawić nowy węzeł na początek.
```

```
        Niestety my musimy iść być może na koniec listy... */

    if (w == NULL)
    {
        // wstaw tu
        w = new wezel;
        w->nast = NULL;
        w->elem = x;
    }
    else if (w->elem == x)
        return; // element już jest - nic nie rób
    else
        insert(x, w->nast); // wstaw gdzieś dalej
}
```

```
void remove(int x, wezel*& w)
{
    if (w == NULL) return; // nic nie rób
    else if (w->elem == x)
    {
        // skasuj się
        wezel* wstary = w;
        w = w->nast;
        delete wstary;

        // koniec - nie idziemy dalej
    }
    else remove(x, w->nast); // szukaj dalej
}
```

```
void print(wezel* w)
{
    cout << "{ ";
    while (w != NULL) {
        cout << w->elem << " ";
        w = w->nast;
    }
    cout << "}" << endl;
}
```

```
void removeAll(wezel*& w)
{
    if (w == NULL) return; // nie ma czego usuwać - koniec

    removeAll(w->nast); // usuń najpierw wszystkie następniki

    delete w; // usuń siebie samego
    w = NULL;
}
```

7.5.3. Implementacja III: drzewo binarne

Jeśli na elementach słownika została określona relacja porządku <=, możemy zaimplementować ten abstrakcyjny typ danych z użyciem drzew binarnych. Dzięki czemu wszystkie trzy podstawowe operacje będą miały *oczekiwaną* złożoność obliczeniową rzędu $O(\log n)$ (por. dyskusję w rozdz. 7.4.2).

Stosunkowo najprostszymi, a zatem i niewymagającymi szerszego komentarza, są operacje *search* i *removeAll*. Dla ilustracji, na rys. 7.10 pokazujemy przebieg wyszukiwania elementu 2 w przykładowym drzewie złożonym z elementów $1, 2, 3, 5, 6, 8$.

```
bool search(int x, wezel* w)
{
   if (w == NULL)          return false;
   else if (w->elem == x) return true;
   else if (w->elem < x)  return search(x, w->prawy);
   else                    return search(x, w->lewy);
}
```

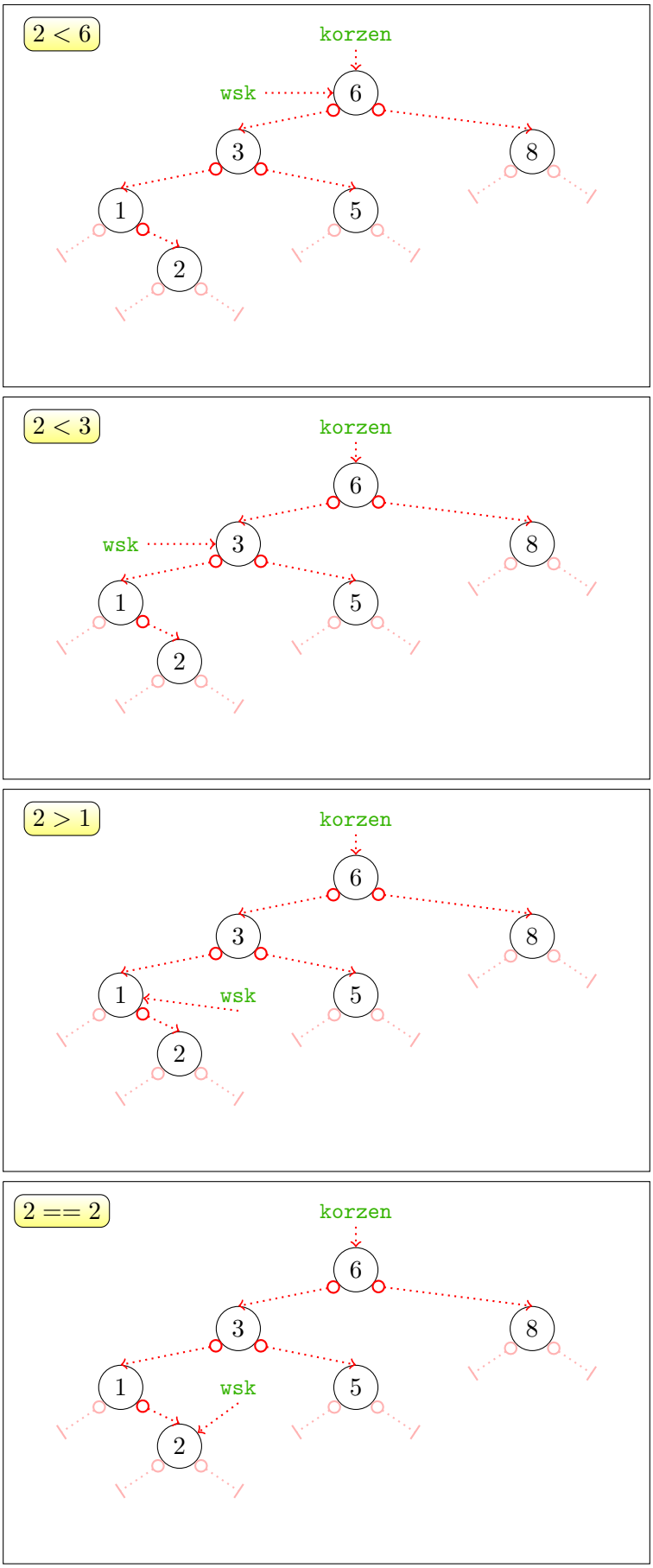
```
void removeAll(wezel*& w)
{
   if (w == NULL) return; // nie ma czego usuwać

   removeAll(w->lewy);    // usuń najpierw
   removeAll(w->prawy);   // wszystkich potomkow

   delete w;     // potem usuń samego siebie
   w = NULL;
}
```

Aby nieco „uatrakcyjnić” sposób wyświetlania drzewa, w operacji *print* skorzystamy z dodatkowej, pomocniczej funkcji rekurencyjnej.

```
void print_pomocnicza(wezel* w) // funkcja pomocnicza
{
   // realizuje ,,zwykłe'' wypisanie elementów
   if (w == NULL) return;
   print_pomocnicza(w->lewy);
   cout << w->elem << " ";
   print_pomocnicza(w->prawy);
}

void print(wezel* w)
{
   cout << "{ ";
   print_pomocnicza(w);
   cout << "}" << endl;
}
```

Operacja *insert* realizowana będzie za pomocą wstawiania nowego elementu jako potomka odpowiednio wybranego liścia.

```
void insert(int x, wezel*& w)
{
   if (w == NULL)
   {                        // wstaw tu
      w = new wezel;
      w->lewy = NULL;
      w->prawy = NULL;
      w->elem = x;
   }
   else if (w->elem == x)
      return;       // element już jest — nic nie rób
   else if (w->elem <  x)
```



Rys. 7.10. Wyszukiwanie elementu 2 w przykładowym drzewie binarnym.

```
      insert(x, w->prawy);
   else
      insert(x, w->lewy);
}
```

Nieco bardziej skomplikowaną będzie ostatnia operacja. Usuwając dany węzeł musimy bowiem rozważyć trzy przypadki:

1. jeśli węzeł jest liściem — po prostu go usuwamy,
2. jeśli jest węzłem z tylko jednym dzieckiem — zastępujemy go jego dzieckiem,
3. jeśli węzeł ma dwoje dzieci — należy go zastąpić wybranym z dzieci, a następnie doczepić drugie dziecko (wraz ze wszystkimi potomkami) za najmniejszym/największym potomkiem pierwszego.

```
void remove(int x, wezel*& w)
{
   if (w == NULL) return;        // nic nie rób
   else if (w->elem < x)
      remove(x, w->prawy);
   else if (w->elem > x)
      remove(x, w->lewy);
   else                          // element znaleziony
   {
      wezel* wstary = w;

      if (w->lewy != NULL)       // ma lewego potomka ---
      {
         w = w->lewy;            // zastąp węzeł lewym dzieckiem

         // A co z ewentualnym prawym potomkiem i jego dziećmi?
         // Musimy go doczepic jako "skrajnie" prawy liść
         wezel* wsk = w;
         while (wsk->prawy != NULL)
            wsk = wsk->prawy;
         wsk->prawy = wstary->prawy;
      }
      else if (w->prawy != NULL) // ma prawego potomka,
                                 // lecz nie ma lewego ---
         w = w->prawy;           // po prostu zastąp prawym
      else                       // jest lisciem ---
         w = NULL;               // po prostu usuń

      delete wstary;
   }
}
```

7.6. Podsumowanie

Przewodnim celem niniejszej części skryptu było przedstawienie czterech bardzo ważnych w algorytmice abstrakcyjnych typów danych. Omówiliśmy:

1. stos (LIFO),
2. kolejkę (FIFO),

3. kolejkę priorytetową (HPF),

4. słownik.

Wiemy już, iż każdy ATD można zaimplementować na wiele sposobów, używając np. tablic albo dynamicznych struktur danych (list, drzew itp.). W każdym przypadku interesowała nas najbardziej wydajna implementacja.

Tablica 7.1 zestawia złożoność obliczeniową najważniejszych operacji wykonywanych na tablicach, listach jednokierunkowych i drzewach binarnych. Każda z tych trzech struktur danych ma swoje „mocne" i „słabe" strony. Z dyskusji przeprowadzonej w poprzednich rozdziałach wiemy już, jak ważny jest świadomy wybór odpowiedniego narzędzia, którym zamierzamy się posłużyć w praktyce.

Tab. 7.1. Złożoność obliczeniowa wybranych operacji.

Operacja	**Tablica**	**Lista**	**Drzewo BST**
Dostęp do i-tego elementu	$O(1)$	$O(n)$	$O(n)$
Wyszukiwanie	$O(n)$ $^{(a)}$	$O(n)$	$O(n)$ $^{(b)}$
Wstawianie	$O(n)$	$O(n)$	$O(n)$ $^{(b)}$
— na początek		$O(1)$	
— na koniec		$O(n)$ $^{(c)}$	
Usuwanie	$O(n)$	$O(n)$	$O(n)$ $^{(b)}$
— z początku		$O(1)$	
— z końca		$O(n)$ $^{(d)}$	

(a) $O(\log n)$, jeśli tablica jest posortowana.
(b) Oczekiwana złożoność $O(\log n)$.
(c) $O(1)$, jeśli lista zawiera wskaźnik na ogon.
(d) Nieomawiana tutaj lista dwukierunkowa pozwala na realizację tej operacji w czasie $O(1)$.

Oczywiście w algorytmice znanych jest o wiele więcej ciekawych algorytmów i struktur danych, np. tablice mieszające (ang. *hash tables*) implementujące słownik. Takowe Czytelnik może poznać studiując bogatą literaturę przedmiotu bądź słuchając bardziej zaawansowanych wykładów.

7.7. Ćwiczenia

Zadanie 7.1. Zaimplementuj w postaci osobnych funkcji następujące operacje na liście jednokierunkowej przechowującej wartości typu `double`:

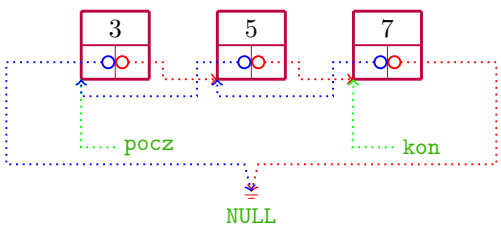
1. Wypisanie wszystkich elementów listy na ekran.
2. Wyznaczenie sumy wartości wszystkich elementów.
3. Wyznaczenie sumy wartości co drugiego elementu.
4. Sprawdzenie, czy dana wartość rzeczywista `y` występuje w liście.
5. Wstawienie elementu na początek listy.
6. Wstawienie elementu na koniec listy.
7. Wstawienie elementu na i-tą pozycję listy.
8. Wstawienie elementu `y` z zachowaniem porządku, tzn. jeśli pierwotna lista jest posortowana, to po wstawieniu porządek nie jest zaburzony.
9. Usuwanie elementu z początku listy. Usuwany element jest zwracany przez funkcję.
10. Usuwanie elementu z końca listy. Usuwany element jest zwracany przez funkcję.
11. Usuwanie i-tego w kolejności elementu.
12. Usuwanie wszystkich elementów o zadanej wartości. Zwracana jest wartość logiczna w zależności od tego, czy choć jeden element znajdował się na liście, czy nie.
13. ⋆ Odwrócenie kolejności elementów listy.
14. Usuwanie z uporządkowanej listy powtarzających się elementów.
15. ⋆ Usuwanie z listy (niekoniecznie uporządkowanej) powtarzających się elementów.
16. ⋆ Przesunięcie co drugiego w kolejności elementu na koniec listy, z zachowaniem ich pierwotnej względnej kolejności.
17. ⋆ Przesunięcie wszystkich elementów o wartości parzystej na koniec listy, z zachowaniem ich pierwotnej względnej kolejności.

Zadanie 7.2. Dla danych dwóch list jednokierunkowych napisz funkcję, która je połączy, np. dla (1,2,5,4) oraz (3,2,5) sprawi, że I lista będzie postaci (1,2,5,4,3,2,5), a II zostanie skasowana.

Zadanie 7.3. Dla danych dwóch *posortowanych* list jednokierunkowych napisz funkcję, która je połączy w taki sposób, że wartości będą nadal posortowane, np. dla (1,2,4,5) oraz (2,3,5) sprawi, że I lista będzie postaci (1,2,2,3,4,5,5), a II zostanie skasowana.

Zadanie 7.4. Rozwiąż zadanie 7.1 zakładając tym razem, że operacje dokonywane są na liście jednokierunkowej, dla której dodatkowo przechowujemy wskaźnik na ostatni element.

Zadanie 7.5. Rozwiąż zadanie 7.1 zakładając tym razem, że operacje dokonywane są na liście dwukierunkowej. Lista dwukierunkowa składa się z węzłów, w których znajduje się nie tylko wskaźnik na następny, ale i także na poprzedni element. Co więcej, prócz wskaźnika na pierwszy element, powinniśmy zapewnić bezpośredni dostęp (np. w funkcji `main()`) do ostatniego, por. zad. 7.4, tzn. dysponujemy jednocześnie „głową” oraz „ogonem” listy. Zaletą listy dwukierunkowej jest możliwość bardzo szybkiego wstawiania i kasowania elementów znajdujących się zarówno na początku jak i na końcu tej dynamicznej struktury danych. Poniższy rysunek przedstawia schemat przykładowej listy dwukierunkowej przechowującej elementy 3, 5, 7.



Zadanie 7.6. Napisz samodzielnie pełny program w języku C++, który implementuje z użyciem listy jednokierunkowej i testuje (w funkcji `main()`) stos (LIFO) zawierający dane typu `double`.

Zadanie 7.7. Napisz samodzielnie pełny program w języku C++, który implementuje z użyciem listy jednokierunkowej z dodatkowym wskaźnikiem na ostatni element i testuje (w funkcji `main()`) zwykłą kolejkę (FIFO) zawierającą dane typu `char*` (napisy).

⋆ **Zadanie 7.8.** Zaimplementuj operacje `enqueue()` i `dequeue()` zwykłej kolejki (FIFO) typu `int` korzystając tylko z dwóch gotowych stosów.

Zadanie 7.9. Napisz samodzielnie pełny program w języku C++, który implementuje i testuje (w funkcji `main()`) kolejkę priorytetową zawierającą dane typu `int`.

Zadanie 7.10. Napisz funkcję, która wykorzysta kolejkę priorytetową do posortowania danej tablicy o elementach typu `int`.

Zadanie 7.11. Zaimplementuj w postaci osobnych funkcji następujące operacje na drzewie binarnym przechowującym wartości typu `double`:

1. Wyszukiwanie danego elementu.
2. Zwrócenie elementu najmniejszego.
3. Zwrócenie elementu największego.
4. Wypisanie wszystkich elementów w kolejności od najmniejszego do największego.
5. Wypisanie wszystkich elementów znajdujących się wierzchołkach będących liśćmi.
6. Sprawdzenie, czy dane drzewo jest zdegenerowane (bezpośrednio redukowalne do listy), tzn. czy każdy element posiada co najwyżej jednego potomka).
7. Wstawianie elementu o zadanej wartości.
8. Wstawianie elementu o zadanej wartości. Jeśli wstawiany element znajduje się już w drzewie nie należy wstawiać jego duplikatu.
9. Usuwanie elementu największego.
10. Usuwanie elementu najmniejszego.
11. Usuwanie elementu o zadanej wartości.
12. Usuwanie co drugiego elementu.
13. Usuwanie wszystkich powtarzających się wartości.
14. ⋆ Odpowiednie przekształcenie drzewa (z zachowaniem własności drzewa binarnego) tak, by element największy znalazł się w korzeniu.
15. ⋆ Wypisanie wszystkich elementów, których wszyscy potomkowie (pośredni i bezpośredni) mają dokładnie dwóch potomków lub są liśćmi.
16. ⋆ Wypisanie wszystkich elementów, których wszyscy przodkowie (pośredni i bezpośredni) przechowują wartości nieparzyste.

Zadanie 7.12. Napisz funkcję, która jako parametr przyjmuje drzewo binarne i zwraca listę jednokierunkową (wskaźnik na pierwszy element) zawierającą wszystkie elementy przechowywane w drzewie uporządkowane niemalejąco.

⋆ **Zadanie 7.13.** Napisz funkcję, która jako parametr przyjmuje drzewo binarne i zwraca listę jednokierunkową (wskaźnik na pierwszy element) zawierającą wszystkie elementy przechowywane w drzewie uporządkowane poziomami (tzn. pierwszy jest korzeń, następnie dzieci korzenia, potem dzieci-dzieci korzenia itd).