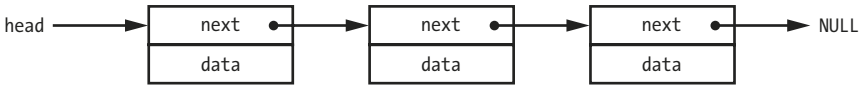
Listy powiązane

Tabele są najbardziej efektywnym sposobem przechowywania list rzeczy. Trzyma się w nich tylko rzeczywiste dane i nie wymagają one żadnych dodatkowych informacji księgowych. Nie działają jednak dobrze dla dowolnych ilości danych, ponieważ jeśli nie zadbaliśmy o to, by tablica była dostatecznie wielka, to trzeba zrobić nową, większą i skopiować wszystkie dane do nowej tablicy. Z drugiej strony, jeśli zrobimy za dużą tablicę, zmarnujemy miejsce. Kopiowanie jest również wymagane, jeśli chcemy włożyć element do środka tabeli, a także w przypadku kasowania elementu.

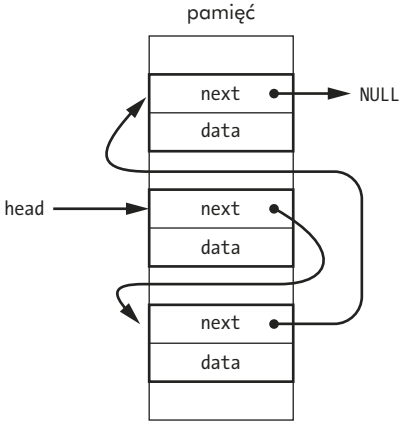
Listy powiązane zachowują się wydajniej w sytuacjach, w których nie wiadomo z góry, ile rzeczy będzie się śledzić. Listy powiązane pojedynczo, implementowane z udziałem struktur, wyglądają tak jak na rysunku 7.10.



Rysunek 7.10. Lista powiązana

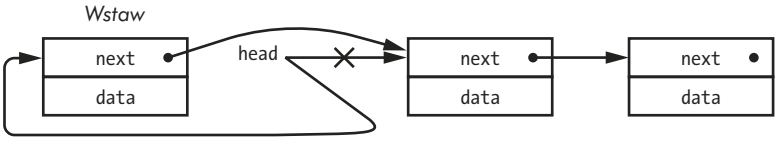
Zauważmy, że `next` jest wskaźnikiem trzymającym adres następnego elementu listy. Pierwszy element listy znany jest jako *głowa* (*head*), ostatni element – jako *ogon* (*tail*). Rozpoznajemy ogon po tym, że jego `next` ma wartość, która nie może być następnym elementem listy. Najczęściej jest to NULL.

Najważniejszą różnicą między listą z rysunku 7.10 a tablicą jest to, że elementy tablicy zawsze zajmują sąsiadujące ze sobą miejsca w pamięci. Elementy listy mogą być rozrzucone po całej pamięci i wyglądać bardziej jak rysunek 7.11.



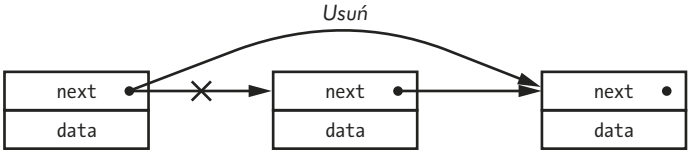
Rysunek 7.11. Lista powiązana w pamięci

Dodanie elementu do listy jest proste; po prostu zróbmy z niego nową głowę, jak to pokazano na rysunku 7.12.



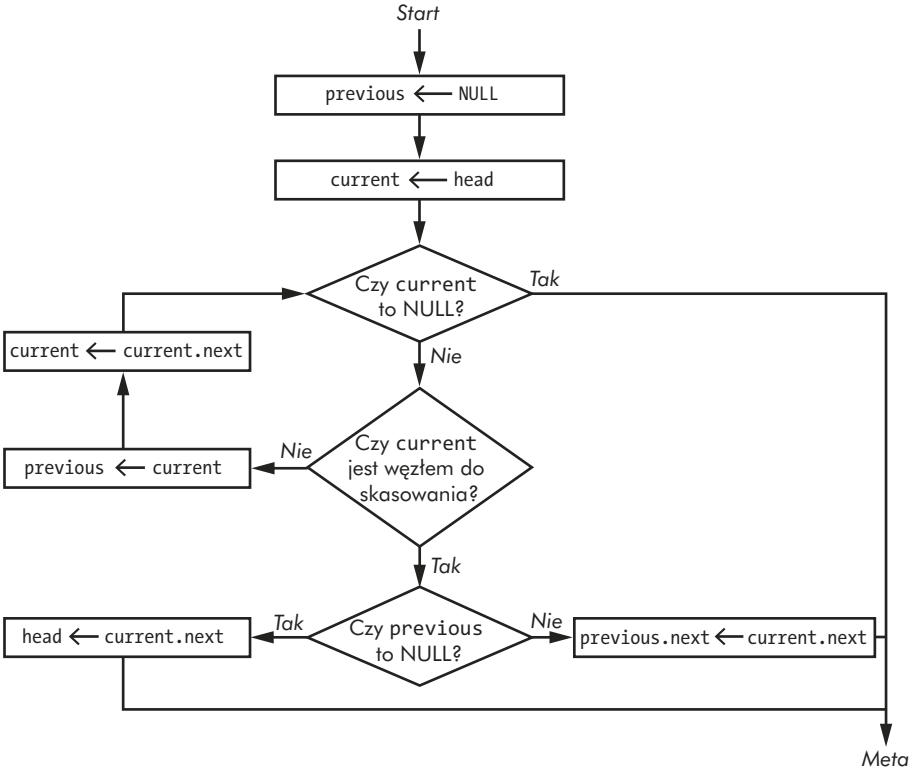
Rysunek 7.12. Wstawianie elementu do listy powiązanej

Kasowanie elementu jest nieco bardziej skomplikowane, ponieważ musimy zadbać, by next poprzedniego elementu wskazywał teraz na następny element – tak jak to pokazano na rysunku 7.13.



Rysunek 7.13. Kasowanie elementu z listy powiązanej

Jednym ze sposobów na to jest użycie pary wskaźników jak na rysunku 7.14.

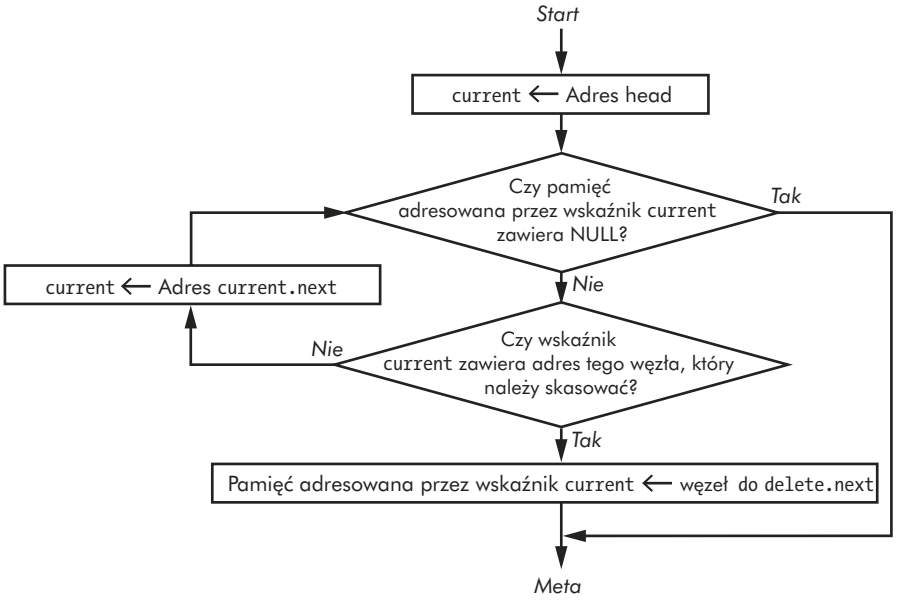


Rysunek 7.14. Kasowanie elementu z listy powiązanej z użyciem pary wskaźników

Wskaźnik current idzie przez listę i poszukuje tego węzła, który trzeba usunąć. Wskaźnik previous pozwala nam dostosować next węzła poprzedzającego ten, który mamy skasować. Używamy kropki (.) do oznaczenia elementu struktury, tak więc current.next oznacza element next węzła current.

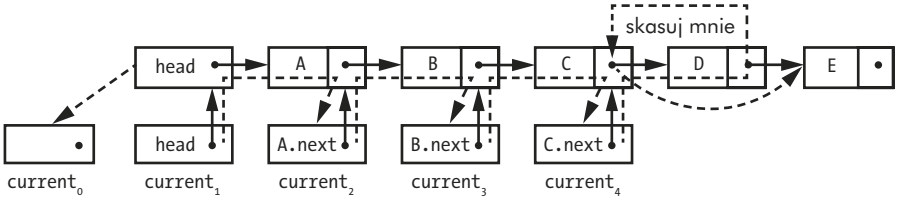
UWAGA *Rysunek 7.14 nie jest świetnym przykładem. Chociaż, mówiąc szczerze, podczas pisania tego punktu zajrzałem do Internetu i znalazłem tam znacznie gorsze algorytmy. Problem z przedstawionym tutaj kodem polega na tym, że jest on skomplikowany, ponieważ potrzebny jest oddzielny test, by rozpoznać głowę listy.*

Algorytm widoczny na rysunku 7.15 pokazuje nam potęgę *podwójnego adresowania pośredniego* (*double indirect addressing*). Eliminuje ono specjalny przypadek, dzięki czemu kod jest prostszy.



Rysunek 7.15. Kasowanie elementu z listy powiązanej za pomocą adresowania pośredniego

Zbadajmy bardziej szczegółowo, co robi ten algorytm. Spójrzmy na rysunek 7.16. Indeksy powiedzą nam, jak zmienia się current w trakcie działania algorytmu.



Rysunek 7.16. Kasowanie elementu z listy powiązanej w akcji

Kroki pokazane na rysunku 7.16 są skomplikowane, więc przejdźmy przez nie kolejno.

1. Zaczynamy od ustawienia $current_0$ na adres zawarty w `head`, co w rezultacie daje $current_1$, wskazujący `head`. To znaczy, że `current` wskazuje `head`, który wskazuje element `A` listy.
2. Element `A` nie jest tym, którego szukamy, więc ruszamy dalej.
3. Jak to pokazuje nam przerywana strzałka, ustawiamy `current` na adres wskaźnika `next` w elemencie wskazanym przez to, co wskazuje `current`. Ponieważ $current_1$ wskazuje `head`, który wskazuje element `A`, $current_2$ będzie wskazywało `A.next`.
4. To nadal nie jest element, którego szukamy, więc robimy to jeszcze raz, w wyniku czego $current_3$ odnosi się do `B.next`.
5. To nadal nie jest element, którego szukamy, więc robimy to ponownie, w wyniku czego $current_4$ odnosi się do `C.next`.
6. `C.next` wskazuje element `D`, który chcemy skasować. Idąc za przerywaną strzałką, podążamy za `current` do `C.next` i za nim do `D`, zamieniając przy tym `C.next` z zawartością `D.next`. Ponieważ `D.next` wskazuje element `E`, `C.next` wskazuje teraz `E` – co zostało na rysunku oznaczone przez następną przerywaną strzałkę. W ten sposób `D` zostało usunięte z listy.

Powyższy algorytm możemy zmodyfikować tak, by wstawiał nowe powiązania w środek listy. Mogłoby się przydać, gdybyśmy chcieli mieć na przykład listę uporządkowaną według daty, nazwy albo jakiegokolwiek innego kryterium.

Wcześniej wspominaliśmy, że drugi algorytm daje nam lepszy kod. Zobaczmy, jak każdy z nich wygląda w języku C i porównajmy je. Nie trzeba koniecznie rozumieć kodu, żeby zobaczyć różnicę między listingiem 7.1 a listingiem 72.

```
struct node {
    struct node *next;
    // data
};

struct node *head;
struct node *node_to_delete;
struct node *current;
struct node *previous;

previous = (struct node *)0;
current = head;

while (current != (struct node *)0) {
    if (current == node_to_delete) {
        if (previous == (struct node *)0)
            head = current->next;
        else
            previous->next = current->next;
        break;
```

```
        }
        else {
                previous = current;
                current = current->next;
        }
    }
```

Listing 7.1. Kod w języku C do kasowania elementu listy powiązanej z użyciem pary wskaźników

```
struct node {
    struct node *next;
    // data
};

struct node *head;
struct node *node_to_delete;
struct node **current;

for (current = &head; *current != (struct node *)0; current = &((*current)->next))
    if (*current == node_to_delete) {
            *current = node_to_delete->next;
            break;
    }
}
```

Listing 7.2. Kod w języku C do kasowania elementu listy powiązanej z użyciem podwójnego adresowania pośredniego

Jak widać, wersja z adresowaniem pośrednim widoczna na listingu 7.2 jest znacznie prostsza od wersji z użyciem pary wskaźników z listingu 7.1.

Dynamiczna alokacja pamięci

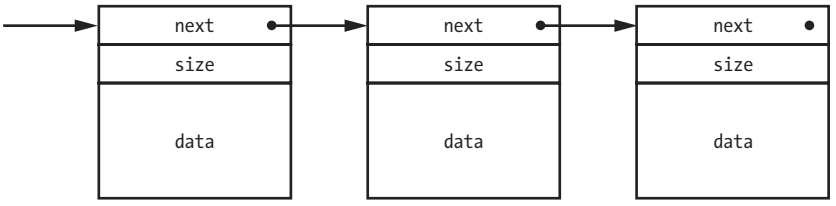
W naszym rozważaniach nad wstawianiem elementu do listu powiązanej wygodnie pominęliśmy coś ważnego. Pokazaliśmy, jak wstawić nowy węzeł, ale nie powiedzieliśmy, skąd wzięliśmy pamięć potrzebną do tego, by ten węzeł mieć.

Widzieliśmy wcześniej na rysunku 5.16, że przestrzeń pamięci programu zaczyna się od sekcji danych alokowanych statycznie, po której następuje sterta ustawiona dla programu przez bibliotekę wykonawczą. Jest to całość pamięci, jaką program ma udostępnioną na swoje dane na maszynach, które nie są wyposażone w jednostki zarządzania pamięcią (MMU). Na systemach z MMU biblioteka wykonawcza wysyła żądanie takiej ilości pamięci, jaka wydaje jej się wystarczająca, ponieważ zajmowanie całej pamięci operacyjnej od razu nie ma najmniejszego sensu. *Break* oznacza koniec pamięci dostępnej dla programu i istnieją pewne wywołania systemowe, które powiększają lub pomniejszają ilość dostępnej pamięci.

Pamięć dla zmiennych takich jak tablica jest statyczna – tzn. przydzielony jej adres nie zmienia się. Takie rzeczy jak węzły listy są dynamiczne – pojawiają się i znikają w miarę potrzeb. Pamięć na nie otrzymujemy ze sterty.

Program potrzebuje jakiegoś sposobu na zarządzanie stertą. Musi wiedzieć, która pamięć jest tam używana, a która dostępna. Istnieją funkcje w bibliotece, które robią dokładnie to, więc nie trzeba pisać swoich. W języku C dysponujemy funkcjami `malloc` i `free`. Zobaczmy, jak można je zaimplementować.

Jedna z możliwych implementacji `malloc` działa z użyciem struktury listy powiązanej. Sterta dzielona jest na bloki, a każdy z tych bloków ma rozmiar i wskaźnik do następnego bloku – tak jak to pokazano na rysunku 7.17.



Rysunek 7.17. Struktura malLoc do zarządzania stertą

Początkowo istnieje tylko jeden blok dla całej sterty. Kiedy program żąda pamięci, `malloc` szuka bloku z dostateczną ilością wolnego miejsca, zwraca wywołującemu wskaźnik do żądanej przestrzeni i dostosowuje rozmiar bloku tak, aby odzwierciedlało to ilość pamięci, którą właśnie przydzielił. Kiedy program zwalnia pamięć za pomocą funkcji `free`, po prostu dodaje zwalniany blok do listy.

Raz na jakiś czas `malloc` przeszukuje listę, by znaleźć sąsiadujące ze sobą nieużywane bloki i jeśli takie zauważy, łączy je w jeden większy blok. Jedną z dobrych okazji, by to zrobić, jest przydzielanie pamięci (za pomocą wywołania `malloc`), ponieważ wymaga to przejścia przez całą listę w poszukiwaniu wolnych bloków. Wraz z upływem czasu pamięć podlega *fragmentacji*, co oznacza, że nigdzie nie ma wolnych bloków o dostatecznie dużych rozmiarach, chociaż nie cała pamięć została wykorzystana. W systemach wyposażonych w MMU dostosowuje się wtedy podział i uzyskuje więcej pamięci, jeśli jest ona potrzebna.

Widać, że zastosowanie tego podejścia kosztuje pewną ilość pamięci: `next` i `size` dodają do każdego bloku 16 bajtów (na 64-bitowej maszynie).

Zwalnianie nieprzydzielonej pamięci to jeden z najczęstszych błędów popełnianych przez niedoświadczonych programistów. Innym takim błędem jest używanie pamięci po tym, jak została zwolniona. Jak widać na rysunku 7.17, jeśli zapiszemy dane poza granicami alokowanej pamięci, możemy uszkodzić pola `size` i `next`. Tego typu błędy są szczególnie podstępne, ponieważ skutki mogą nie pojawić się przez długi czas – widoczne są dopiero wtedy, gdy chcemy uzyskać jakąś informację z uszkodzonych pól.

Jeden ze skutków ubocznych postępu technologii jest taki, że małe maszyny często mają o wiele więcej RAM-u, niż potrzeba naszym programom. W takich przypadkach najlepiej jest po prostu statycznie alokować całość dostępnej pamięci, ponieważ to zmniejsza wielkość „haraczu”, jaki musimy płacić i eliminuje niebezpieczeństwo błędów związanych z alokacją pamięci.