Uniwersytet Zielonogórski
Instytut Sterowania i Systemów Informatycznych

Algorytmy i struktury danych

Laboratorium Nr 3

Algorytmy sortowania wewnętrznego

1 Cel ćwiczenia

Ćwiczenie ma na celu zapoznanie studentów z wybranymi metodami sortowania wewnętrznego. Są to: sortowanie przez proste wstawianie, sortowanie przez proste wybieranie (przez selekcję), sortowanie bąbelkowe, sortowanie szybkie (*quicksort*), sortowanie stogowe i sortowanie przez scalanie.

2 Wstęp

Metoda sortowania jest nazywana **wewnętrzną**, jeśli dane uporządkowane są w postaci tablicy (zatem mieszczą się w pamięci operacyjnej RAM) oraz w procesie sortowania nie korzysta się z tablic pomocniczych.

Dobra metoda sortowania powinna charakteryzować się **stabilnością**. Metoda sortowania jest stabilna, jeśli zachowuje względną kolejność elementów ze zdublowanymi kluczami. Przykładowo, jeśli ułożoną alfabetycznie listę osób posortuje się po roku urodzenia, metoda stabilna da w efekcie listę, w której osoby urodzone w tym samym roku będą nadal ułożone w kolejności alfabetycznej, natomiast metoda niestabilna nie gwarantuje tego.

3 Metody sortowania wewnętrznego

3.1 Sortowanie przez proste wstawianie

Metodę tę można porównać do porządkowania kart na ręku: należy brać kolejne, jeszcze nie uporządkowane karty, i wstawiać na odpowiednie miejsce pomiędzy karty już uporządkowane.

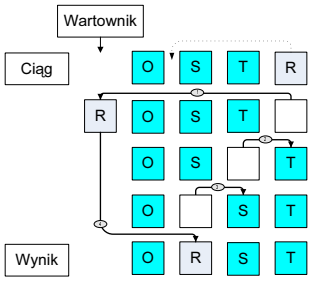
Zakładając, że ciąg do posortowania oznaczony jest przez a i składa się on z N elementów, sortowanie przez proste wstawianie odbywa się w następujący sposób: dla każdego $i = 2, 3, \ldots, N$ trzeba powtarzać wstawianie elementu a_i w już uporządkowaną część ciągu $a_1 \leq \cdots \leq a_{i-1}$. W metodzie tej obiekty podzielone są umownie na dwa ciągi: ciąg wynikowy $a_1, a_2, \ldots, a_{i-1}$ oraz ciąg źródłowy $a_i \ldots a_n$. W każdym kroku, zaczynając od $i = 2$, i-ty element ciągu źródłowego przenoszony jest do ciągu wynikowego i wstawiany w odpowiednie miejsce, po czym wartość i jest inkrementowana. Pętla zaczyna się od $i = 2$, gdyż zakładamy, że element a_1 już znajduje się na właściwym miejscu.

Ponieważ przy przenoszeniu elementów w lewo istnieje możliwość przekroczenia zakresu tablicy, najczęściej stosuje się **wartownika**. Dla uproszczenia kodu zakłada się, że wartownik jest umieszczany pod indeksem 0. Wartownik musi być równy przenoszonemu elementowi, co zapewni, że nie da się wyjść poza tablicę.

Wstawienie elementu w odpowiednie miejsce ciągu ilustruje rys. 1. Kolorem jasnoniebieskim zaznaczono elementy już posortowane, niebieskim - element do wstawienia.

W zaprezentowanym przykładzie wartownikiem jest element a_0. Pełni on także rolę bufora, w którym przechowywany jest aktualny element na czas przesunięcia innych elementów. Nie można wykroczyć się poza pierwszy element tego ciągu, gdyż element a_0 ciągu, będący wartownikiem, jest zawsze elementem mniejszym lub równym od wstawianego, więc pętla zawsze zakończy się prawidłowo.

Przykład opisuje wstawienie litery R pomiędzy elementy O i S. W pierwszym kroku kopiujemy literę R na miejsce a_0, i staje się ona wartownikiem. Po skopiowaniu miejsce a_4, w którym dotychczas była litera R, możemy potraktować jako „wolne” i wpisać tam inny element ciągu. Kopiujemy zatem literę a_3



Rys. 1: Wstawianie elementu do uporządkowanej tablicy

(litera T) do a_4 - teraz „wolne” miejsce to a_3. Powtarzając kopiowanie do momentu natrafienia na literę większą lub równą przenoszonej, czyli jak w przykładzie literze O, otrzymamy wolne miejsce, w które należy wstawić przenoszoną literę R.

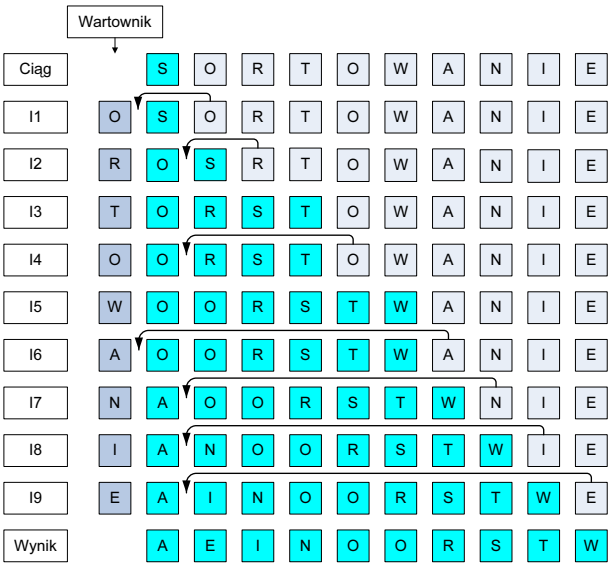
```
Wstaw element a_i na miejsce o indeksie 0, otrzymując wartownika;
Dopóki element a_0 jest mniejszy od a_{i-1} wykonuj
    przypisz elementowi a_i element a_{i-1}
    zmniejsz i
Przypisz elementowi a_i element a_0
```

Przypisujemy do zmiennej a_0 element a_i, następnie dopóki $a_0 < a_i$ wpisujemy do i-tego elementu ciągu element $i-1$, zmniejszając jednocześnie i. Na zakończenie elementowi i-temu trzeba przypisać wartość a_0.

Algorytm sortowania przez proste wstawianie działa stosunkowo wolno, jest on bowiem algorytmem klasy $\mathcal{O}(N^2)$. Oznacza to, że w praktyce algorytm ten nie jest stosowany do sortowania długich ciągów. Jednocześnie zaletą algorytmu jest jego prostota.

Aby posortować całą tablicę, należy po prostu wykonać algorytm wstawiania elementów do uporządkowanej tablicy dla elementów od indeksu 2 do N.

Rozpatrzmy działanie algorytmu na przykładowym ciągu liter `SORTOWANIE`. Na rys. 2, który obrazuje tę operację, elementy posortowane oznaczono kolorem jasnoniebieskim.



Rys. 2: Sortowanie przez proste wstawianie

Na początku rozważana jest druga litera ciągu - litera O (pierwsza litera ciągu - litera S - znajduje się już na właściwym miejscu). Litera O zostaje skopiowana na miejsce wartownika (a_0). Następnie powinna ona zostać wstawiona na odpowiednie miejsce w ciągu, czyli należy przenieść ją przed literę S. Warto zauważyć, że w tej iteracji przed wyjściem z lewej strony poza zakres tablicy chroni wartownik. W iteracji nr 3 na właściwym miejscu, pomiędzy literami O i S, zostaje umieszczona litera R.

W kolejnych iteracjach należy postępować analogicznie. W niektórych iteracjach (np iteracja nr 3) przestawienie nie jest w ogóle konieczne, w pozostałych na właściwych miejscach zostają umieszczone odpowiednie litery.

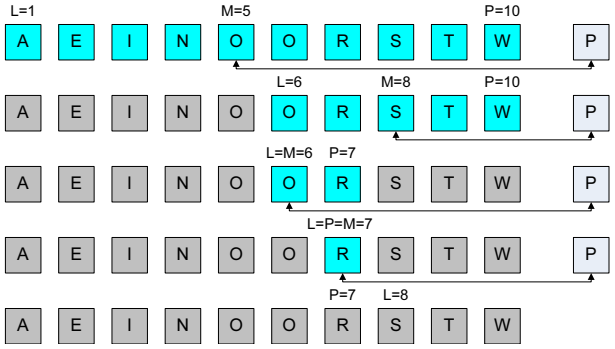
3.1.1 Sortowanie przez wstawianie połówkowe

Ta metoda sortowania jest ulepszeniem sortowania przez proste wstawianie. Modyfikacja opiera się na prostej obserwacji: ponieważ wstawiamy elementy do posortowanej tablicy, można łatwo znaleźć miejsce, w które należy wstawić element dzięki metodzie przeszukiwania binarnego. Poza tą zmianą algorytm działa na identycznej zasadzie, jak zwykłe sortowanie przez proste wstawianie.

Dzięki takiej modyfikacji zmniejsza się ilość porównań, liczba przesunięć pozostaje niezmieniona. Ponieważ jednak przesuwanie sortowanych elementów jest zwykle kosztowniejsze niż ich porównywanie, usprawnienie to nie jest znaczące.

Dany jest ciąg o elementach $a_l \dots a_p$ oraz element do wstawienia el. Poszukujemy miejsca w ciągu, w które powinien zostać wstawiony element el za pomocą algorytmu przeszukiwania binarnego. Można zapisać ten algorytm w następującej postaci:

```
Dopóki l ≤ p wykonuj:
    Oblicz indeks elementu środkowego m ze wzoru m = ⌊ (l + p)/2 ⌋;
    Jeśli element eljest mniejszy od elementu a_m:
        Odetnij prawą część tablicy przypisując p = m − 1;
    W przeciwnym wypadku:
        Odetnij lewą część tablicy, przypisując l = m + 1;
```



Rys. 3: Przeszukiwanie binarne

Poszukujemy miejsca, w które należy wstawić literę P w ciągu $AEINOORSTW$ (indeksowany od 1 do 10). Na początku granice przeszukiwania określone są następująco - lewa $l = 1$, a prawa $p = 10$, gdyż przeszukujemy cały ciąg. Szukamy indeksu środkowego elementu ciągu - jest to $m = \lfloor (l + p)/2 \rfloor = 5$, porównujemy zatem literę P ze środkową literą ciągu (litera O). Ponieważ litera P jest większa od O, a wiadomo, że ciąg jest uporządkowany, to możemy odrzucić lewą część ciągu (na lewo od litery O włącznie z samą literą O). Robimy to poprzez przesunięcie lewego indeksu poszukiwań - teraz do l należy wstawić $m + 1 = 6$. Na rysunku odrzucona część tablicy wyróżniona jest kolorem szarym.

Następnie czynności powtarzamy: szukamy środka ciągu wg opisanej wyżej metody, otrzymując 8. Ze sprawdzenia ($P < S$) wynika, że tym razem litera w tablicy jest większa od szukanej, a więc można przesunąć prawą granicę przeszukania na $m - 1 = 7$. Kontynuując w ten sposób do momentu, gdy lewa granica jest mniejsza lub równa od prawej, otrzymamy $l = 8$ i $p = 7$. Ponieważ $L \geq R$, przeszukiwanie kończy się. Indeks L jest miejscem, w które należy wstawić literę P.

3.2 Sortowanie przez proste wybieranie (przez selekcję)

Jest to kolejny prosty algorytm, który można opisać w następujący sposób: należy znaleźć najmniejszy element ciągu N-elementowego i zamienić go miejscami z pierwszym elementem tego ciągu. Następnie znaleźć najmniejszy element w ciągu $N - 1$-elementowym (pierwszy element jest już posortowany, więc szukać najmniejszego elementu należy od indeksu 2) i zamienić go miejscami z drugim elementem ciągu. Kontynuując w ten sam sposób, ciąg zostanie posortowany.

Oznaczając ciąg symbolem a i zakładając, że ma on N elementów, sortowanie przez wybieranie polega na wyznaczeniu najmniejszego elementu ciągu $a_i \dots a_N$ dla każdego $i = 1, 2, \dots, N-1$ i zamienieniu go elementem a_i. Operację tę powtarza się do czasu posortowania całego ciągu.

W większości implementacji liczba zamian w tym algorytmie wynosi $N-1$, liczba porównań natomiast jest proporcjonalna do N^2. Cechy te predestynują sortowanie przez proste wybieranie do obróbki zbiorów z dużymi polami danych i małymi kluczami, gdyż w takich zastosowaniach koszt przemieszczania danych dominuje nad kosztem porównań, a żaden inny algorytm nie dokonuje mniejszej ilości przestawień podczas sortowania.

Wadą algorytmu sortowania przez wybieranie jest fakt, iż czas jego działania praktycznie nie zależy od stopnia uporządkowania zbioru sortowanego - oznacza to, że algorytm ten sortuje już posortowany ciąg lub ciąg zawierający te same liczby w takim samym czasie, jak ciąg całkowicie losowy.

Rys. 4: Sortowanie przez proste wybieranie

Sam algorytm ma następującą postać:

```
Wykonuj co następuje od indeksu i = 1 do i = N − 1:
    Wskaż najmniejszy element spośród a_i ... a_N;
    Zamień ten element z elementem a_i.
```

Przebieg sortowania ciągu `SORTOWANIE` obrazuje rys. 4. Ciąg posortowany oznaczony jest kolorem niebieskim.

W pierwszym przebiegu pętli najmniejszym elementem jest A, co zostało zaznaczone kolorem ciemnoniebieskim. Element ten jest zamieniany z pierwszym elementem ciągu (litera S). W drugiej iteracji najmniejsze jest E, dlatego też jest ono zamieniane z drugim elementem ciągu (pierwszym nieposortowanym) - literą O.

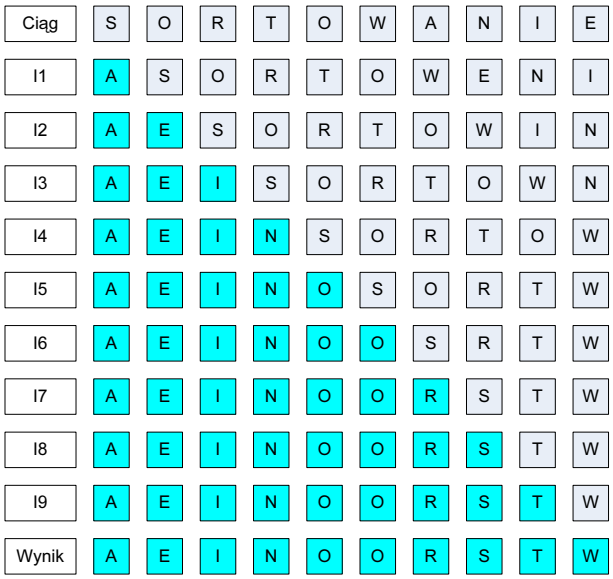
W każdej kolejnej iteracji wyznaczane jest minimalny element w nieposortowanej części ciągu (elementy już posortowane oznaczane są kolorem jasnoniebieskim). Następnie, jeśli element minimalny nie znajduje się na właściwym miejscu, zamienia się go z pierwszym elementem nieposortowanym. Zdarza się, że element najmniejszy znajduje się już w ciągu posortowanym (tak, jak np w iteracji 5) - w takim przebiegu pętli nie zmienia się kolejności elementów w ciągu.

3.3 Sortowanie bąbelkowe

Metoda ta polega na zamienianiu miejscami dwóch sąsiadujących ze sobą elementów do czasu uzyskania zbioru uporządkowanego. Nazwa algorytmu pochodzi od analogii do pęcherzyków powietrza, ulatujących w górę tuby wypełnionej wodą. Zakładamy, że ciąg "przeczesywany" będzie zawsze od prawej do lewej strony. Trafiając na najmniejszy element ciągu, zamieniany on jest z kolejnymi elementami aż trafi na pierwsze miejsce ciągu. W drugim przebiegu postępuje się podobnie z drugim najmniejszym elementem, co prowadzi do umieszczenia tego elementu na drugim miejscu. Kontynuując otrzyma się ciąg posortowany.

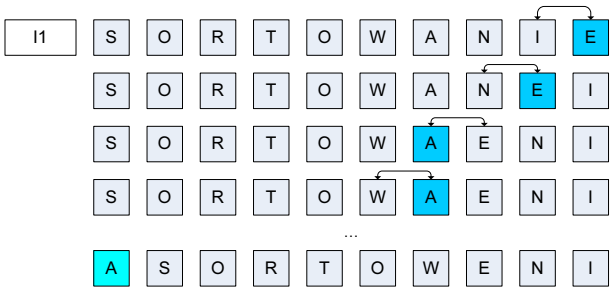
Algorytm ma następującą postać:

```
Wykonuj co następuje N − 1 razy od indeksu i = N − 1 do i = 1:
    Wskaż na ostatni element;
    Wykonaj co następuje i razy od indeksu j = 1:
        Porównaj element j-ty z elementem j + 1;
        Jeśli porównywane elementy są w niewłaściwej kolejności
                (element j  >  j + 1), zamień je miejscami.
```



Rys. 5: Sortowanie bąbelkowe

Rozważmy działanie algorytmu na przykładowym ciągu $SORTOWANIE$ (rys. 5).



Rys. 6: Sortowanie bąbelkowe - pierwsza iteracja

W pierwszej iteracji (rys. 6) ostatnia litera E jest zamieniana kolejno z literami I oraz N (kolorem ciemnoniebieskim zaznaczono litery mniejsze - jeśli litera mniejsza znajduje się na dalszym miejscu ciągu, konieczna jest jej zamiana z literą poprzedzającą); zamiana zatrzymuje się na literze A. Następnie to litera A będzie zamieniana z każdą literą ją poprzedzającą i przesunie się na początek ciągu (litera A jest najmniejsza w ciągu).

W kolejnych iteracjach, przy analogicznym postępowaniu, pojedyncze litery trafiają na właściwe miejsce w ciągu.

Główną zaletą metody sortowania bąbelkowego jest łatwość jej implementacji. Algorytm ten ma natomiast wiele wad: zdarzają się "puste przebiegi" ciągu bez zamian, bo elementy są już posortowane, dodatkowo algorytm ten jest bardzo wrażliwy na konfigurację danych. Przykładowo, poniższe ciągi, prawie nie różniące się od siebie, wymagają różnej ilości zamian: w pierwszym przypadku jednej, a w drugim aż sześciu:

Ciąg A	4	2	6	18	20	39	40
Ciąg B	4	6	18	20	39	40	2

3.4 Sortowanie szybkie

Algorytm sortowania szybkiego (*quicksort*) jest najczęściej używanym algorytmem spośród wszystkich algorytmów sortowania. Podstawowa wersja tego algorytmu została wynaleziona w 1960 r. przez C.A.R. Hoare'a. Popularność algorytmu sortowania szybkiego jest spowodowana jego zaletami:

- do posortowania n elementów wymaga średnio czasu proporcjonalnego do $\mathcal{O}(n \log n)$;
- czas działania algorytmu dla rzeczywistych przypadków jest najczęściej zbliżony do czasu średniego, a w wielu przypadkach czas ten jest nawet liniowy;
- jest dość prosty w implementacji, biorąc pod uwagę szybkość działania.

Algorytm ten ma również wady:

- jest niestabilny, tzn. nie ma gwarancji zachowania kolejności takich samych elementów w tablicy posortowanej; np mając tablicę rekordów zawierających nazwiska oraz wiek, posortowaną po nazwiskach, sortując tę tablicę po wieku nie ma gwarancji, że osoby w tym samym wieku będą umieszczone alfabetycznie w tablicy posortowanej;
- pesymistyczna złożoność tego algorytmu wynosi $\mathcal{O}(n^2)$.

Algorytm sortowania szybkiego jest przykładem wykorzystania techniki "dziel-i-rządź". Technika ta polega na wykorzystaniu cechy rekurencji: dekompozycji problemu na pewną ilość skończonych podproblemów tego samego typu, a następnie połączeniu otrzymanych częściowych rozwiązań w celu otrzymania rozwiązania globalnego. W wielu przypadkach technika ta pozwala na zmianę klasy algorytmu (np z $\mathcal{O}(n^2)$ do $\mathcal{O}(n \log_2 n)$).

Algorytm sortowania szybkiego działa na zasadzie podziału ciągu na dwie części i niezależnego, rekurencyjnego sortowania każdej z tych części. Najważniejszą częścią algorytmu jest proces podziału, który działa następująco:

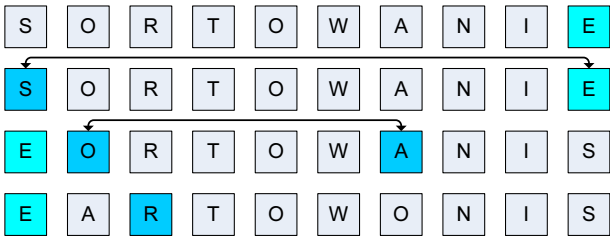
- Wybierany jest **element osiowy** - może to być dowolny element ciągu;
- Dokonuje się zamiany elementów ciągu tak, aby uzyskać dwie podtablice, pierwsza o elementach mniejszych lub równych elementowi osiowemu i druga o elementach większych lub równych elementowi osiowemu.

Po dokonaniu podziału następuje rekurencyjne wywołanie algorytmu osobno dla części lewej i prawej. Można łatwo wykazać przez indukcję, że takie postępowanie prowadzi do posortowania danych, gdyż zawsze w wyniku podziału przynajmniej jeden element (osiowy) trafia na właściwą pozycję.

Przykładowy schemat postępowania przy podziale ciągu jest następujący:

- dokonujemy wyboru elementu osiowego. Zwykle wybiera się pierwszy lub ostatni element ciągu;
- przeglądamy ciąg od jego lewego końca do momentu znalezienia elementu większego lub równego elementowi osiowemu;
- przeglądamy ciąg od prawego końca, do czasu znalezienia elementu mniejszego lub równego elementowi osiowemu;
- jeśli indeksy powyższych przeszukiwań nie minęły się, zamieniamy miejscami oba znalezione elementy;
- kontynuujemy przeglądanie ciągu z lewej i prawej strony oraz zamianę miejscami elementów do czasu, gdy indeksy miną się.

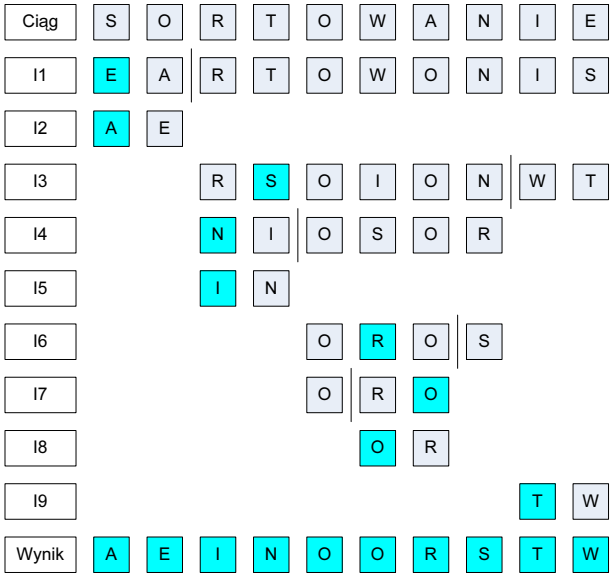
Przykład działania procesu podziału obrazuje rys. 7. Zakłada się, że elementem osiowym jest ostatni element ciągu (w przykładzie jest to litera E), co zostało na rysunku oznaczone kolorem jasnoniebieskim. Idąc od początku ciągu szukamy pierwszego elementu większego lub równego od osiowego - warunek ten spełnia litera S. Następnie idąc od prawej strony szukamy elementów mniejszych lub równych od osiowego - jak łatwo się domyślić, trafiamy na sam element osiowy, czyli literę E. Zamieniamy miejscami litery S oraz E. Kontynuując, z lewej strony elementem większym od osiowego jest O, natomiast z prawej strony jest to dopiero litera A (litery N oraz I są większe od elementu osiowego - litery E). Zamieniamy miejscami litery O oraz A. W kolejnej iteracji z lewej strony dochodzimy do elementu R, a z prawej - do elementu A, który jednak leży przed elementem R. Oznacza to zakończenie „przeczesywania" tablicy.



Rys. 7: Sortowanie szybkie - proces podziału

Po dokonaniu podziału ciągu na dwa podciągi, wystarczy tylko wywołać rekurencyjnie funkcję wykonującą szybkie sortowanie dla lewego oraz dla prawego podciągu. Podział na podciągi przebiega w miejscu minięcia się indeksów (w przykładzie - na literze R, czyli dla indeksu równego 3).

Przykład działania algorytmu szybkiego sortowania dla ciągu `SORTOWANIE` obrazuje rys. 8. Poszczególne linie przedstawiają postać ciągu już po wykonaniu podziału (zakłada się, że element osiowy jest ostatnim elementem ciągu - w każdej iteracji został on oznaczony kolorem jasnoniebieskim), miejsce podziału oznaczono pionową kreską.



Rys. 8: Sortowanie szybkie

W pierwszej iteracji elementem osiowym jest E. Po dokonaniu podziału, wywoływana jest rekurencyjnie procedura sortująca na części lewej (ciąg EA) oraz prawej (ciąg $RTOWONIS$). Dalsza część algorytmu wykonywana jest na identycznej zasadzie.

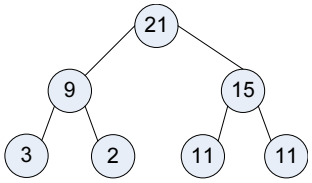
Podczas implementowania algorytmu sortowania szybkiego należy pamiętać o:

- zapewnieniu zakończenia programu rekurencyjnego, czyli:
  - procedura nie może wywołać sama siebie dla ciągów o długości 1 lub mniejszej
  - procedura będzie wywoływać sama siebie dla ciągów o liczności mniejszej niż ilość elementów, z którymi została ona wywołana.
- Częstym błędzie implementacji polegającym na zapomnieniu o tym, by za każdym razem przynajmniej jeden element był wstawiany na właściwą pozycję. Błąd taki kończy się nieskończoną pętlą rekurencyjną, gdy element osiowy okazuje się być najmniejszym lub największym elementem ciągu.

3.5 Sortowanie stogowe

Aby omówić sortowanie stogowe, niezbędne jest wprowadzenie kilku podstawowych pojęć.

Drzewo binarne jest drzewem, dla którego każdy ojciec ma co najwyżej dwóch bezpośrednich synów. Strukturę taką przedstawia rys. 9. Każdy element drzewa, posiadający elementy dołączone na niższym



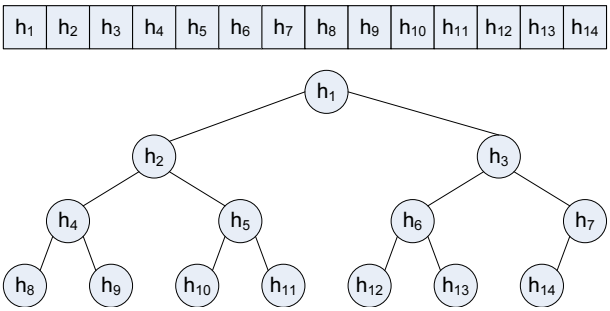
Rys. 9: Przykładowe drzewo binarne, uporządkowane stogowo

poziomie, jest ich **poprzednikiem** (ojcem), natomiast same elementy dołączone są jego **następnikami** (zwanymi również potomkami lub synami). Na przykład, na rys. 9 elementy „9" i „15" są bezpośrednimi następnikami (synami) elementu „21". Element „9" jest także bezpośrednim poprzednikiem (ojcem) elementów „3" i „2". **Korzeń** drzewa to element, nie posiadający poprzedników (znajdujący się na samej górze struktury - w przykładzie to element „21").

W drzewie uporządkowanym stogowo każdy poprzednik (ojciec) jest większy lub równy od wszystkich swoich następników (synów). Oznacza to, że korzeń drzewa jest jego największym elementem. Drzewo z rys. 9 jest uporządkowane stogowo.

Stóg jest zbiorem elementów, ułożonych w pełne drzewo binarne, uporządkowane stogowo i przedstawione w formie tablicy. Można przyjąć następującą zasadę: element na pozycji i ma następniki na pozycjach $2 * i$ oraz $2 * i + 1$; analogicznie element na pozycji i ma poprzednika na pozycji $\lfloor i/2 \rfloor$. Następniki elementu są mniejsze od samego elementu. Pośrednio można wnioskować, że na podstawie tej zasady żaden węzeł drzewa uporządkowanego stogowo nie jest większy od korzenia tego drzewa. A zatem ciąg kluczy: $h_l, h_{l+1}, \ldots, h_p$ jest stogiem, jeśli $h_i \leq h_{2*i} \wedge h_i \leq h_{2*i+1}$.

Tablicowa reprezentacja drzewiasta tablicy H może przedstawiać się następująco:



Rys. 10: Tablica H oraz drzewo binarne

Rozpoczynając od pierwszego elementu tablicy do jej ostatniego elementu, dla wybranego elementu o indeksie i traktowanego jako poprzednik definujemy dwa następniki jako elementy o indeksach $2 * i$ i $2 * i + 1$, np. dla elementu h tablicy H o indeksie 1 (h_1) jego następnikami będą elementy tablicy H o indeksach 2 (h_2) i 3 (h_3). Analogicznie dla elementu h_{13} jego następnikami, zgodnie z opisaną zależnością, będą elementy h_{26} oraz h_{27}. Dla każdego następnika można równie łatwo znaleźć jego poprzednika: w tablicy wystarczy odnaleźć element o indeksie $\lfloor i/2 \rfloor$ - np dla elementu h_5 jego poprzednikiem będzie element h_2. Korzeniem drzewa jest element h_1 (o indeksie 1).

Oczywiście muszą zachodzić zależności stogu, tzn. dla każdego i-tego elementu tablicy jego następniki na pozycjach $2 * i$ oraz $2 * i + 1$ muszą być mniejsze od tego elementu.

Należy zwrócić uwagę, że w podanym przykładzie element h_7 ma tylko jednego następnika - h_{14}. Wynika to z faktu, że w tablicy H jest tylko 14 elementów. Mamy tutaj 7 poprzedników i jednocześnie 13 następników, czyli elementy $h_1, h_2, \ldots, h_7$ są poprzednikami (są elementami z przynajmniej jednym następnikiem); jednocześnie elementy $h_2, h_3, \ldots, h_{13}$ są następnikami (są elementami z przypisanym poprzednikiem). A zatem w stogu część elementów jest jednocześnie poprzednikiem i następnikiem.

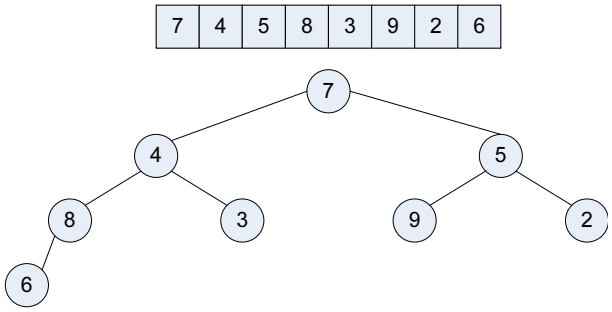
3.5.1 Algorytm tworzenia stogu z tablicy - przesiewanie stogu

Jak wspomniano, stóg ma właściwość taką, że każdy i-ty element jest większy od swoich następników (elementów o indeksach $2 * i$ oraz $2 * i + 1$). Zatem aby sprawdzić, czy tablica jest uporządkowana stogowo, należy przeanalizować wszystkie elementy będące poprzednikami i sprawdzić, czy ich następniki są mniejsze od tych elementów, a jeśli nie, zamienić miejscami badany element z następnikiem, który

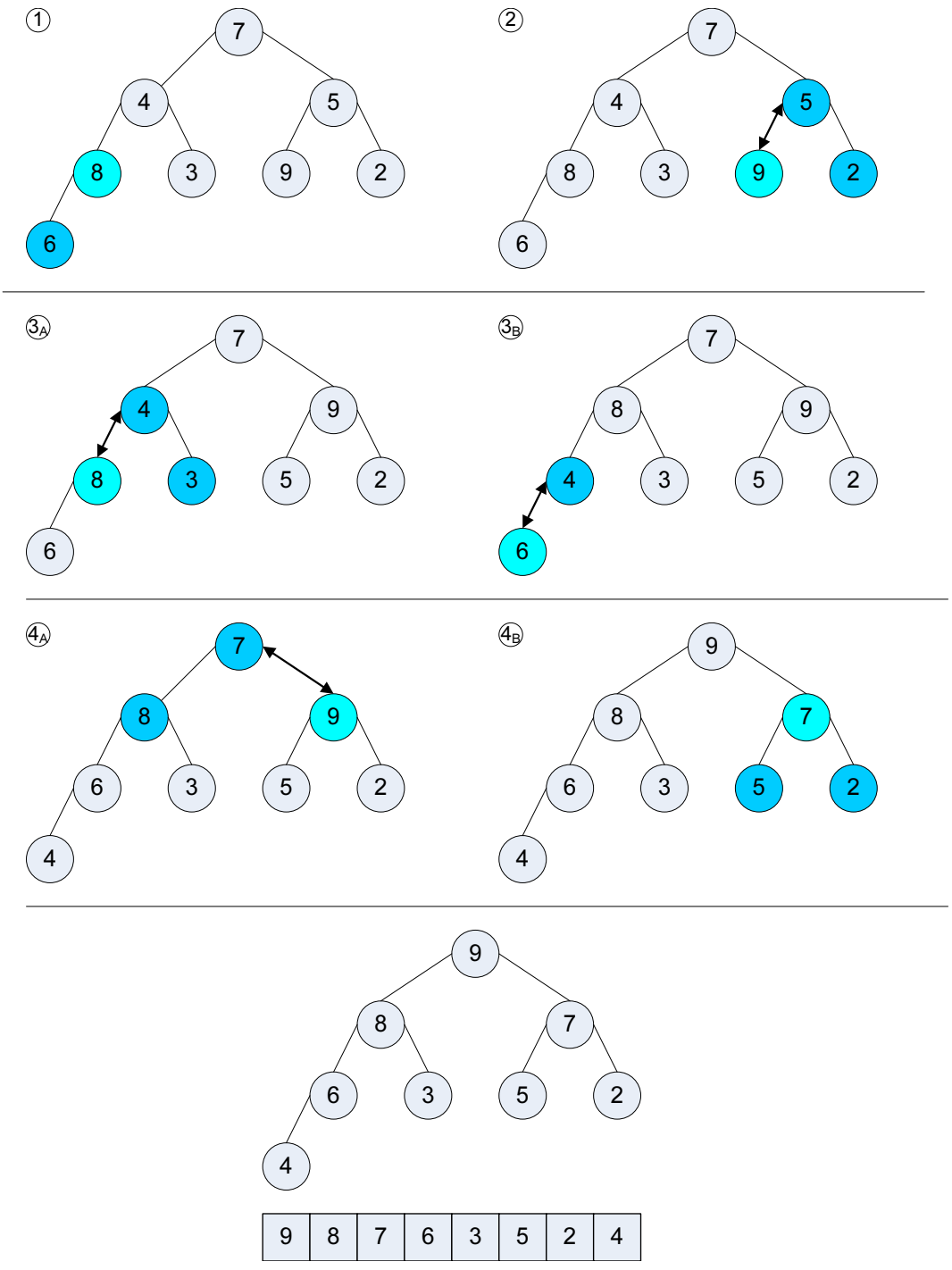
jest od niego większy. Po dokonaniu zamiany należy jeszcze sprawdzić, czy przesiany w dół element nie zakłócił porządku stogowego. Aby zmniejszyć ilość porównań, porównuje się następniki elementu, a sam element porównuje się z większym z jego następników.

Przykład działania procedury przesiewania

Dana jest tablica oraz reprezentowane przez nią drzewo binarne:



Rys. 11: Tablica H oraz reprezentowane przez nią drzewo binarne



Rys. 12: Działanie procedury przesiewania

Teraz zaczynając od ostatniego elementu, będącego poprzednikiem wprowadzamy porządek stogowy w drzewie (rys. 12).

- Krok 1. Obliczamy indeks ostatniego elementu będącego poprzednikiem poprzez obliczenie ($m = \lfloor N/2 \rfloor$, gdzie N jest rozmiarem tablicy) - jest to wartość 4, a zatem indeks wskazuje na element 8. Element ten ma jeden następnik, który jest mniejszy od niego, a zatem pierwsze „poddrzewo", którego korzeniem jest element 4, spełnia własności stogu.
- Krok 2. Przechodzimy do elementu równego 5 o indeksie 3. Ten element ma dwóch następników, większym z nich jest 9 - a zatem poddrzewo nie jest stogiem (gdyż potomek 9 jest większy od swojego poprzednika) Aby ustanowić porządek stogowy, zamieniamy miejscami element 5 z elementem 9. Ponieważ żaden z następników nowego korzenia poddrzewa (elementu 9) nie jest ma swoich następników, po dokonaniu zamiany całe poddrzewo jest stogiem.
- Krok 3_A. Element równy 4 o indeksie 2. Ten element ma dwóch potomków - większym z nich jest 8, więc poddrzewo również nie jest stogiem (element 8 jest większy od poprzednika, elementu 4). Dokonujemy zatem zamiany miejscami elementu 4 i 8.
- Krok 3_B. Teraz należy sprawdzić, czy całe poddrzewo jest już uporządkowane stogowo, sprawdzamy zatem, czy zamieniony element 4 jest większy od swoich następników (ma tylko jednego). Okazuje się, że nie - więc konieczna jest kolejna zamiana elementów (element 4 o nowym indeksie 4 z elementem 6 o indeksie 8). Dopiero po przeczesaniu całego poddrzewa można uznać, że spełnia ono cechy stogu.
- Krok 4_A. Porównujemy element równy 1 o indeksie 1 z większym z jego następników (elementy o indeksach 2 i 3) i dokonujemy niezbędnej zamiany (element 3 z elementem 1).
- Krok 4_B. Sprawdzamy poddrzewo, którego korzeniem jest zamieniony moment wcześniej element 2. Zamiana nie jest wymagana, drzewo spełnia warunek stogu.
- Krok 5. Kończymy przesiewanie - całe drzewo jest uporządkowane stogowo. Otrzymujemy tablicę, przedstawioną w dolnej części rys. 12.

Sam algorytm ma postać:

```
Wykonuj co następuje od indeksu i = ⌊ N/2 ⌋ do i = 1:
    Wskaż na i-ty element tablicy;
    Wybierz większego z następników tego elementu (jeśli ma jednego następnika, wybierz go);
    Jeśli następnik jest większy od badanego elementu wykonuj
        Zamień miejscami badany element i-ty z jego większym potomkiem;
        Przypisz do i indeks elementu, który został przesiany w dół.
```

3.5.2 Sortowanie stogowe

Sortowanie stogowe polega na wielokrotnym ($n-1$-krotnym, gdzie n oznacza rozmiar tablicy do posortowania) wykonaniu następującej sekwencji poleceń (zakładamy, że w tablicy jest już zaprowadzony porządek stogowy):

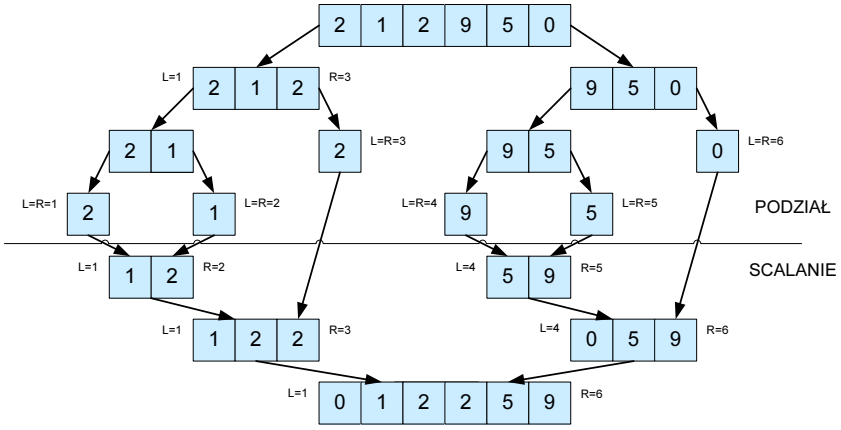
```
Wykonuj co następuje od indeksu i =  N  do i = 2:
    Zamień i-ty element drzewa z korzeniem tego drzewa (elementem nr 1);
    Przesiej drzewo, zawierające elementy od 1 do i − 1;
```

3.6 Sortowanie przez scalanie

Działanie algorytmu sortowania przez scalanie można podzielić na dwie fazy: najpierw następuje podział tablicy na dwie równe części (lub prawie równe, gdy ilość elementów w tablicy jest nieparzysta), następnie każda z tych części jest nadal dzielona na pół itd. Proces podziału kończy się w momencie, gdy przetwarzana część tablicy zawiera tylko jeden element. Wtedy rozpoczyna się proces scalania poszczególnych fragmentów tablicy. Zasadę działania algorytmu sortowania przez scalanie przedstawia rys. 13.

Uwaga! W algorytmie sortowania przez scalanie, zgodnie z definicją algorytmu sortowania wewnętrznego, **nie wolno używać dodatkowej tablicy!**.

Zakładając, że sortowana tablica zawiera N elementów, algorytm sortowania można przedstawić następująco:



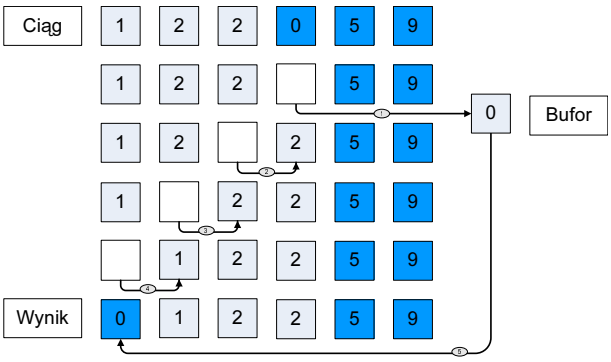
Rys. 13: Sortowanie przez scalanie

```
Wykonuj co następuje dopóki tablica ma więcej niż jeden element:
    Podziel tablicę na dwie (prawie) równe części;
    Wywołaj rekurencyjnie funkcję sortującą na lewej części tablicy;
    Wywołaj rekurencyjnie funkcję sortującą na prawej części tablicy;
    Dokonaj scalenia obu części tablicy.
```

A zatem ciąg jest dzielony w każdym kroku na dwa niemal równoliczne podciągi, które rekurencyjnie porządkowane są tą sama metodą. Warunkiem zakończenia rekurencji w tym algorytmie jest sytuacja, kiedy ciąg ma tylko jeden element. Zatem powrót z wywołań rekurencyjnych rozpoczyna się z ciągami złożonymi z pojedynczych elementów, które są następnie scalane w ciągi o długości dwa, a następnie w ciągi o długości trzy lub cztery elementy itd.

Sam proces sortowania wymaga pamiętania lewej i prawej granicy „podtablic", na które ciąg jest dzielony. Należy pamiętać, że podział na „podtablice" jest umowny, a wszystkie dane cały czas znajdują się w jednej tablicy, więc granice oznaczają tylko położenie „podtablic" w tablicy danych. Granice wszystkich „podtablic" zostały oznaczone na rys. 13, np w drugiej linii są dwie „podtablice": 212 oraz 950, ich granicami są odpowiednio 1 i 3 oraz 4 i 6.

Po zakończeniu rekurencyjnego dzielenia tablic, rozpoczyna się proces scalania. Analizując ten proces łatwo zauważyć, że zarówno lewa, jak i prawa część scalanej tablicy są zawsze posortowane prawidłowo, wystarczy więc elementy z prawej części tablicy „wkomponować" w część lewą. Proces ten przypomina nieco sortowanie przez proste wstawianie, jednakże w tej sytuacji nie ma wartownika, należy więc kontrolować, czy podczas zamiany elementów indeksy cały czas wskazują na elementy tablicy. Fragment procesu scalania dwóch „podtablic" przedstawia rys. 14. Elementy drugiego ciągu są wyróżnione kolorem ciemnoniebieskim. Rysunek prezentuje wstawienie na właściwe miejsce tylko pierwszej cyfry drugiego ciągu (0), z kolejnymi cyframi drugiego ciągu należy postępować analogicznie.



Rys. 14: Proces scalania dwóch ciągów - dla pierwszego elementu drugiego ciągu

Algorytm scalania dwóch podtablic (l i r oznaczają odpowiednio lewy i prawy koniec tablicy) można zapisać następująco:

```
Wykonuj co następuje od indeksu i = ⌈(l + r)/2⌉ do końca tablicy
```

```
Skopiuj i − ty element tablicy do bufora tymczasowego;
Ustaw indeks j na elemencie i − 1;
Dopóki j-ty element tablicy jest większy od bufora tymczasowego wykonuj:
    Przenieś element j-ty na miejsce j + 1;
    Zmniejsz j o 1;
Wstaw zawartość bufora tymczasowego w j + 1 miejsce tablicy.
```

4 Zadania do wykonania

1. Podać wynik działania każdego algorytmu dla ciągu `PRZYKLADSORTOWANIA`.
2. Które z poznanych algorytmów sortowania są stabilne?

Literatura

[1] L.Banachowski i in. *Algorytmy i struktury danych*; WNT, 1996.

[2] T.H.Cormen i in. *Wprowadzenie do algorytmów*, WNT, 2000

[3] A.Drozdek *Struktury danych w jezyku C*, WNT, 1996

[4] D.E.Knuth *Sztuka programowania*, WNT, 2002

[5] R.Sedgewick *Algorytmy w C++*, Wydawnictwo RM, 1999

[6] P.Wróblewski, *Algorytmy, struktury danych i techniki programowania*, HELION, 1996

[7] N. Wirth, *Algorytmy + struktury danych = programy*, WNT, 2001