Podstawy Programowania 2

Algorytmy Quicksort i Heapsort

Arkadiusz Chrobot

Zakład Informatyki

28 maja 2019

Plan

Wstęp

Dzisiejszy wykład będzie dotyczył dwóch algorytmów sortowania, które powiązane są z wcześniej poruszanymi tematami: metodą „dziel i zwyciężaj”, rekurencją oraz drzewami. Pierwszym z nich jest algorytm Quicksort, drugim algorytm sortowania z użyciem kopca (ang. *Heapsort*).

Wstęp

Zanim przejdziemy do opisu wspomnianych algorytmów przedstawimy funkcje i inne elementy kodu źródłowego, które będą wspólne dla wszystkich zaprezentowanych na tym wykładzie programów.

Wstęp

Pliki nagłówkowe i typ tablicowy

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4
5  typedef int int_array_type[10];
```

Wstęp

Funkcja pliki nagłówkowe i typ tablicowy

Wyniki działania prezentowanych programów będą przedstawiane na ekranie komputera, a do tworzenia danych wejściowych zostanie zastosowany generator liczb pseudolosowych, dlatego włączane są zaprezentowane na poprzednim slajdzie pliki nagłówkowe. Na potrzeby programów został również zdefiniowany typ tablicowy określający tablicę 10 elementów typu `int` (wiersz nr 5).

Wstęp

Funkcja `fill_array()`

```
1  void fill_array(int_array_type array)
2  {
3      int i;
4      srand(time(0));
5      for(i=0;i<sizeof(int_array_type)/sizeof(array[0]);i++)
6          array[i] = -10+rand()%21;
7  }
```

Wstęp

Funkcja `fill_array()`

Zadaniem zaprezentowanej na poprzednim slajdzie funkcji jest wypełnienie tablicy liczbami całkowitymi wylosowanymi z zakresu do -10 do 10. Tablica do wypełnienia jest przekazywana przez parametr i jest typu `int_array_type`, który został zdefiniowany na początku programu. Dzięki temu możemy określić liczbę jej elementów, użytą do określenia zakończenia pętli `for` stosując wyrażenie dzielące rozmiar typu tablicowego przez rozmiar pierwszego elementu tablicy tego typu (wiersz nr 5).

Wstęp

Funkcja print_array()

```
1  void print_array(int_array_type array)
2  {
3      int i;
4      for(i=0;i<sizeof(int_array_type)/sizeof(*array);i++)
5          printf("%d ",array[i]);
6      printf("\n");
7  }
```

Wstęp

Funkcja `print_array()`

Funkcja `print_array()` wypisuje zawartość przekazanej jej przez parametr `array` tablicy na ekran i przenosi kursor do następnego wiersza na ekranie. Od innych funkcji wykonujących podobną czynność w programach zaprezentowanych na wcześniejszych wykładach różni ją kilka szczegółów. Pierwszym jest to, że typ parametru jest wcześniej zdefiniowany w programie za pomocą `typedef`, drugim jest to, że po zakończeniu pętli `for` do przeniesienia kursora do następnego wiersza ekranu jest używana funkcja `printf()`, a argumentem jej wywołania jest ciąg znaków składający się ze znaku nowego wiersza (wiersz nr 6). Ostatnim szczegółem jest wyrażenie użyte do określenia zakończenia pętli `for`. Jest ono podobne do tego, które jest użyte w funkcji `fill_array()`, jednak dostęp do pierwszego elementu tablicy odbywa się za pomocą wskaźnika, a nie notacji z nawiasami kwadratowymi (wiersz nr 4).

Wstęp

Funkcja swap()

```
1  void swap(int *first, int *second)
2  {
3      int temporary = *first;
4      *first = *second;
5      *second = temporary;
6  }
```

Wstęp

Funkcja `swap()`

Kod źródłowy funkcji `swap()` był opisywany na wykładach wielokrotnie, więc teraz tylko przypominamy, że służy ona do zamiany wartości między dwiema zmiennymi. W przypadku programów z dzisiejszego wykładu będą to elementy tablicy.

Quicksort

Algorytm Quicksort został opracowany przez brytyjskiego informatyka C.A.R. Hoare'a i jest jednym z najszybszych algorytmów sortowania w przypadku typowych zastosowań. Wprawdzie w przypadku pesymistycznym jego złożoność obliczeniowa wynosi $\Theta(n^2)$, gdzie n jest liczbą elementów tablicy, ale w przypadku średnim i optymistycznym jest ona równa $\Theta(n \cdot log_2(n))$, przy czym stałe ukryte w tej notacji są małe. Quicksort sortuje tablicę w miejscu, ale jego złożoność przestrzenna wynosi $O(n)$. Wynika to z faktu, że algorytm ten jest rekurencyjny i jest implementowany z użyciem rekursji, stąd zużywa pamięć na stosie. Można go zaimplementować w sposób iteracyjny, ale jest to dosyć złożona czynność, a efektywność wersji iteracyjnej nie jest większa od efektywności wersji rekurencyjnej. Sortowanie wykonywane przez Quicksort jest sortowaniem niestabilnym.

Metoda „dziel i zwyciężaj” w Quicksort

Ida algorytmu Quicksort może być zaprezentowana z użyciem metody „dziel i zywcięża j”:

Dziel: Tablica $A[p \ldots r]$ jest dzielona (wartości jej elementów są zamieniane miejscami) na dwa niepuste obszary $A[p \ldots q]$ i $A[q{+}1 \ldots r]$ takie, że wartość każdego elementu z $A[p \ldots q]$ nie jest większa od wartości dowolnego elementu z $A[q + 1 \ldots r]$. Indeks q jest wyznaczany przez podprogram dzielący.

Zwyciężaj: Dwa obszary $A[p \ldots q]$ i $A[q + 1 \ldots r]$ są sortowane za pomocą rekurencyjnych wywołań algorytmu Quicksort.

Połącz: Ponieważ obszary są sortowane w miejscu, to nie trzeba podejmować żadnych dodatkowych działań, aby je połączyć: cała tablica $A[p \ldots r]$ jest już posortowana.

Quicksort

Funkcja quicksort()

```
1  void quicksort(int_array_type array, int low, int high)
2  {
3      if(low<high) {
4          int partition_index = partition(array,low,high);
5          quicksort(array, low, partition_index);
6          quicksort(array, partition_index+1, high);
7      }
8  }
```

Quicksort

Funkcja `quicksort()`

Funkcja `quicksort()` dopowiada krokowi **Zwyciężaj** w zaprezentowanej wcześniej analizie algorytmu z użyciem metody „dziel i zwyciężaj”. Nie zwraca ona żadnej wartości, ale posiada trzy parametry. Przez pierwszy przekazywana jest tablica do posortowania, a przez drugi i trzeci parametr indeksy wyznaczające obszar tej tablicy, który ma być uporządkowany. Początkowo, np. przy wywołaniu `quicksort()` z poziomu funkcji `main()`, obszar ten obejmuje całą tablicę. Wewnątrz `quicksort()` najpierw sprawdzane jest, czy wartość indeksu początkowego obszaru (`low`) jest mniejsza od wartości indeksu końcowego obszaru (`high`) jeśli tak, to oznacza to, że jest jeszcze obszar tablicy, który trzeba będzie uporządkować, w przeciwnym przypadku funkcja zakończy swoje działanie. Jeśli warunek z wiersza nr 3 jest spełniony, to uruchamiana jest funkcja `partition()`, która porządkuje zadany obszar tablicy i wyznacza punkt podziału tego obszaru na dwa mniejsze. Następnie funkcja `quicksort()` jest wywoływana dwukrotnie, dla każdego z obszarów z osobna.

Quicksort

Funkcja `quicksort()`

Pierwszy obszar porządkowany przez rekurencyjne wywołanie `quicksort()` obejmuje elementy tablicy, których indeksy zawarte są między indeksem początkowym (`low`) i indeksem podziału (`partition_index`). Oba indeksy są włączone do tego zakresu. Drugi obszar obejmuje elementy tablicy o indeksach w zakresie od indeksu podziału (ale bez niego) do indeksu końcowego (`high`), włącznie. Funkcja `partition()` jest w programie zdefiniowana przed `quicksort()`, ale ze względu na czytelność prezentacji zostanie opisana po niej.

Quicksort

Funkcja partition()

```
1  int partition(int_array_type array, int low, int high)
2  {
3      int pivot = array[low];
4      int i = low-1, j = high+1;
5
6      while(i<j) {
7          while(array[--j]>pivot)
8              ;
9          while(array[++i]<pivot)
10             ;
11         if(i<j)
12             swap(&array[i],&array[j]);
13     }
14     return j;
15 }
```

Quicksort

Funkcja `partition()`

Funkcja `partition()` odpowiada krokowi **Podziel** z analizy algorytmu metodą „dziel i zwyciężaj”. Posiada ona trzy parametry o takim samym znaczeniu, jak parametry funkcji `quicksort()`, a zwraca liczbę, która jest indeksem wyznaczającym punkt podziału bieżąco porządkowanego obszaru tablicy na dwa mniejsze. W trzecim wierszu funkcji jest deklarowana i inicjowana zmienna o nazwie `pivot`, która zawiera tzw. wartość osiującą, czyli tą, według której będzie porządkowany dany obszar tablicy. Jako ta wartość przyjmowana jest wartość pierwszego elementu tablicy należącego do bieżąco porządkowanego obszaru (wiersz nr 3). W wierszu nr 4 deklarowane i inicjowane są zmienne, które będą używane do indeksowania porządkowanego obszaru od jego początku (zmienna `i`) i od końca (zmienna `j`). Proszę zwrócić uwagę, że początkowo oba te indeksy określają elementy znajdujące się *poza* porządkowanym obszarem.

Quicksort

Funkcja `partition()`

Zewnętrzna pętla `while` (wiersz nr 6) jest wykonywana tak długo, jak długo wartość indeksu `i` jest mniejsza od wartości indeksu `j`, innymi słowy tak długo aż te indeksy się nie „pokryją” lub nie „miną”. Wewnątrz tej pętli wykonywane są dwie kolejne pętle `while`. Pierwsza (wiersze nr 7 i 8) przeszukuje zadany obszar od końca ku początkowi szukając elementu, którego wartość będzie mniejsza lub równa wartości osiującej. Druga (wiersze nr 9 i 10) przeszukuje ten sam obszar w odwrotnym kierunku szukając elementu, którego wartość będzie większa lub równa wartości osiującej. Proszę zwrócić uwagę na konstrukcję tych pętli. Cała czynność przeszukiwania odbywa się w warunku pętli. Zastosowano w nich predekrementację i preinkrementację indeksów, aby uniknąć sięgania do elementów tablicy znajdujących się poza porządkowanym obszarem, a w niektórych przypadkach poza tablicą.

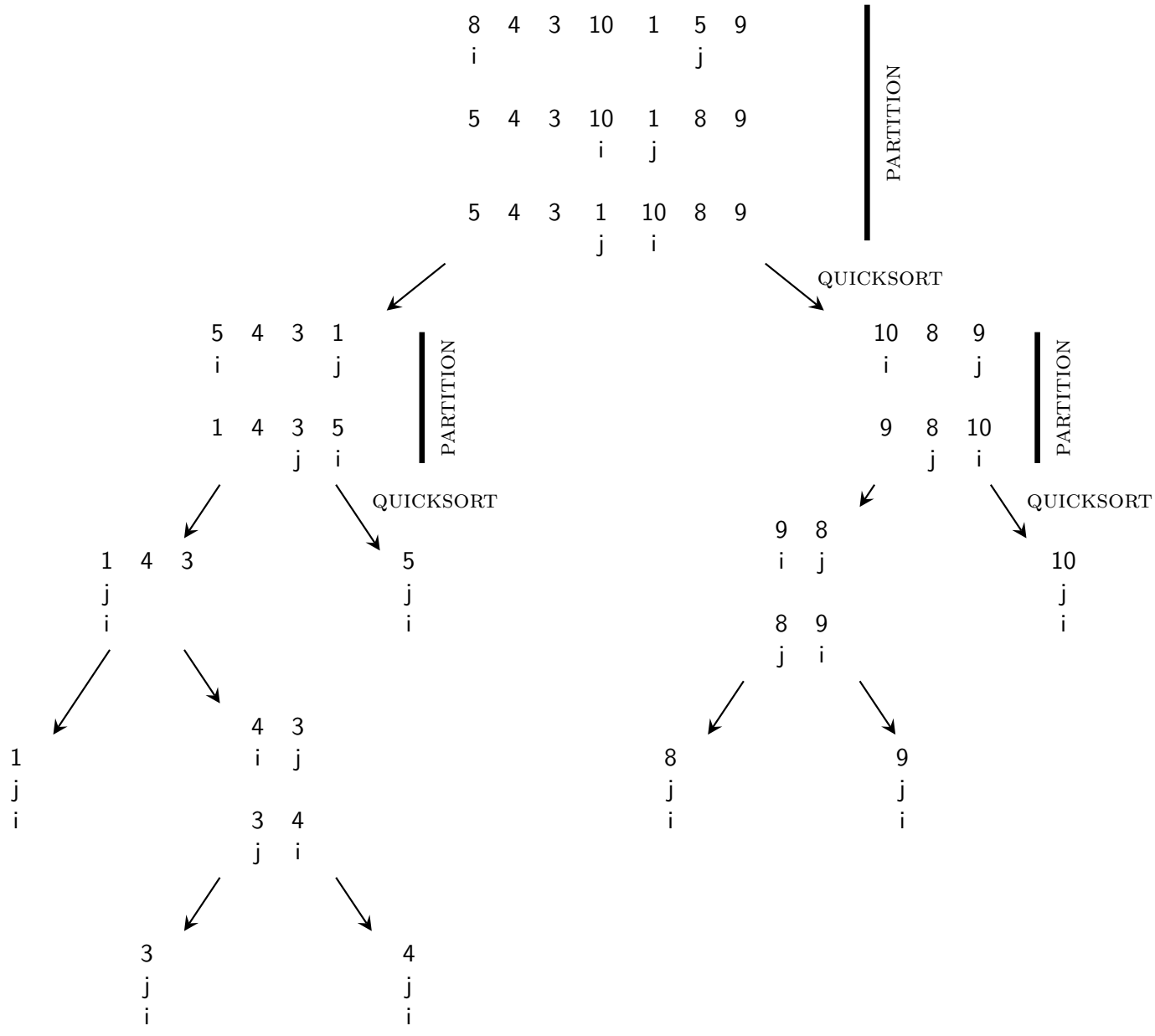
Quicksort

Funkcja `partition()`

Po zakończeniu obu pętli wewnętrznych w instrukcji warunkowej (wiersz nr 11) funkcja sprawdza, czy wartość indeksu `i` jest mniejsza od wartości indeksu `j`. Jeśli tak, to należy zamienić kolejność wartości elementów, które one określają, bo nie jest ona prawidłowa. Jeśli nie, to zewnętrzna pętla `while` się zakończy, a indeks `j` będzie określał punkt podziału dla porządkowanego obszaru tablicy, dlatego jego wartość jest zwracana w wierszu nr 14.

Następny slajd zawiera drzewo ilustrujące działanie funkcji `quicksort()` dla nieposortowanej tablicy liczb naturalnych, o siedmiu elementach. W górnej części drzewa zaznaczono, który fragment jest wykonywany w funkcji `patition()`, a który w `quicksort()`. W dolnej nie zastosowano tego rozwiązania, aby nie zmniejszać czytelności drzewa.

Quicksort


8 4 3 10 1 5 9
i j
5 4 3 10 1 8 9
i j
5 4 3 1 10 8 9
j i
PARTITION
QUICKSORT
5 4 3 1
i j
1 4 3 5
j i
PARTITION
QUICKSORT
10 8 9
i j
9 8 10
j i
PARTITION
QUICKSORT
1 4 3
j i
5
j i
9 8
i j
8 9
j i
10
j i
1
j i
4 3
i j
3 4
j i
8
j i
9
j i
3
j i
4
j i

Funkcja main()

```
1  int main(void)
2  {
3      int_array_type array;
4      fill_array(array);
5      print_array(array);
6      quicksort(array,0,
7          sizeof(int_array_type)/sizeof(*array)-1);
8      print_array(array);
9      return 0;
10 }
```

Funkcja `main()`

W funkcji `main()` została zdefiniowana tablica typu `int_array_type` (wiersz nr 3), która następnie jest wypełniana przy pomocy funkcji `fill_array()` liczbami całkowitymi, wypisywana na ekranie przy pomocy `print_array()` i sortowana z użyciem funkcji `quicksort()`. Ponieważ jako argumenty tej funkcji muszą być podane, oprócz tablicy, jej indeksy, to jako drugi argument do tej funkcji przekazywana jest wartość `0` (indeks pierwszego elementu). Jako trzeci argument jest przekazywana wielkość tablicy wyliczona dzięki takiemu samemu wyrażeniu, jak w funkcji `print_array()`, pomniejszona o jeden. Posortowana tablica jest wypisywana na ekranie i funkcja `main()` kończy swoje działanie zwracając wartość `0`.

Inne wersje

Istnieje kilka wersji algorytmu Quicksort, które w różny sposób są implementowane. Kolejne slajdy zawierają kody źródłowe innej popularnej implementacji funkcji `quicksort()` i `partition()`. Nie jest ona tak efektywna, jak ta przedstawiona wcześniej, ale niektórzy informatycy uważają ją za bardziej czytelną i łatwiejszą do stworzenia, a zatem dającą mniej okazji do popełnienia błędów.

Quicksort

Funkcja quicksort() wersja druga

```
1  void quicksort(int_array_type array, int low, int high)
2  {
3      if(low<high) {
4          int partition_index = partition(array,low,high);
5          quicksort(array, low, partition_index-1);
6          quicksort(array, partition_index+1, high);
7      }
8  }
```

Quicksort

Funkcja `quicksort()` wersja druga

W przypadku funkcji `quicksort()` zaprezentowanej na poprzednim slajdzie jest tylko jeden szczegół, który różni ją od jej poprzedniczki zaprezentowanej wcześniej. Obszar, który jest porządkowany przez jej pierwsze wywołanie rekurencyjne nie zawiera elementu tablicy określanego indeksem `partition_index` (wiersz nr 5).

Quicksort

Funkcja partition() wersja druga

```
1  int partition(int_array_type array, int low, int high)
2  {
3      int pivot = array[low], middle = low, i;
4
5      for(i=low+1; i<=high; i++)
6          if(array[i]<pivot) {
7              middle++;
8              swap(&array[middle],&array[i]);
9          }
10     swap(&array[low],&array[middle]);
11     return middle;
12 }
```

Quicksort

Funkcja `partition()` wersja druga

W tej implementacji funkcji `partition()` użyta jest tylko jedna pętla i jest to pętla `for`. Podobnie jak poprzednio najpierw deklarowanych jest kilka zmiennych lokalnych. Zmienna `pivot` ma takie samo znaczenie, jak w poprzedniej wersji. Zmienna `middle` to indeks elementu, który powinien mieć ostatecznie element w porządkowanym obszarze, który będzie zawierał wartość osiującą. Zmienna `i` jest indeksem pętli. Idea tej implementacji funkcji `partition()` jest następująca: należy tak przestawić wartości w elementach porządkowanego obszaru, aby wartości mniejsze od osiującej znalazły się po jej lewej stronie, a większe po jej prawej. Skoro przyjmujemy za wartość osiującą wartość pierwszego elementu tego obszaru, to należy najpierw założyć, że wszystkie pozostałe jego elementy mają wartość większą lub równą osiującej. Jeśli tak nie jest, to należy ich wartości zamienić miejscami. Wykonywane jest to w pozostałej części funkcji.

Quicksort

Funkcja `partition()` wersja druga

Proszę zwrócić uwagę na pętlę `for`. Iteruje ona po wszystkich elementach porządkowanego obszaru, poza pierwszym. Zatem indeks `i` jest inicjowany wartością o jeden większą od wartości parametru `low`, a pętla kończona jest dopiero wtedy, gdy jego wartość przekroczy wartość parametru `high`. W pętli, w instrukcji warunkowej (wiersz nr 6) sprawdzane jest, czy bieżący element obszaru tablicy ma wartość mniejszą od wartości osiującej. Jeśli tak jest, to zwiększana jest wartość indeksu `middle` i wartość elementu przez niego określanego jest zamieniana miejscami z wartością elementu określanego przez indeks `i`. Po zakończeniu pętli należy tylko wartość osiującą przemieścić we właściwe miejsce obszaru, tzn. z pierwszego jego elementu do elementu określanego przez indeks `middle`. Dokonywane jest to w 10 wierszu funkcji. Po jego wykonaniu zwracana jest wartość indeksu `middle`, która stanowi punkt podziału obszaru.

Quicksort

Funkcja `qsort()`

Algorytm Quicksort jest na tyle efektywny, że twórcy języka C postanowili umieścić w standardowej bibliotece funkcję `qsort()`, która go implementuje. Jej prototyp znajduje się w pliku nagłówkowym `stdlib.h`. Funkcja ta nie zwraca żadnej wartości, ale posiada cztery parametry. Przez pierwszy, który jest wskaźnikiem typu `void *` przekazywana jest tablica do posortowania, przez drugi przekazywana jest liczba elementów tej tablicy, a przez trzeci rozmiar pojedynczego elementu. Czwarty parametr to wskaźnik na funkcję porównującą wartości elementów tablicy, która powinna mieć następujący prototyp:

```
int compare(const void *, const void *);
```

Przez parametry do tej funkcji przekazywane są wskaźniki na porównywane elementy tablicy. Jeśli wartość pierwszego jest mniejsza od drugiego, to funkcja powinna zwrócić liczbę ujemną, jeśli większa to dodatnią, a jeśli są równe, to `0`. Taki sposób konstrukcji `qsort()` pozwala jej sortować tablice elementów dowolnego typu.

Quicksort

Funkcja `qsort()`

Kolejne slajdy zawierają definicję funkcji porównującej dla elementów tablicy typu `int` oraz ilustrują sposób wywołania `qsort()` dla tej tablicy.

Quicksort

Funkcja compare_int()

```
1  int compare_int(const void *first, const void *second)
2  {
3      return *(int *)first - *(int *)second;
4  }
```

Quicksort

Funkcja `compare_int()`

Definicja tej funkcji jest krótka. Opisywana funkcja ma dwa parametry wskaźnikowe o nazwach `first` i `second`. Jej treść stanowi w zasadzie jeden wiersz, w którym oba wskaźniki są najpierw rzutowane na typ wskaźnikowy `int *`, a następnie wykonywana jest ich dereferencja i odejmowane są od siebie wskazywane przez nie wartości. Jeśli wynik będzie ujemny, to pierwsza jest mniejsza od drugiej, dodatni - zachodzi między nimi odwrotna relacja, a zero oznacza, że są one sobie równe. Ten wynik jest zwracany i funkcja kończy swe działanie.

Quicksort

Funkcja `main()`

```
1  int main(void)
2  {
3      int_array_type array;
4      fill_array(array);
5      print_array(array);
6      qsort((void*)array,
7          sizeof(int_array_type)/sizeof(array[0]),
8          sizeof(array[0]),compare_int);
9      print_array(array);
10     return 0;
11 }
```

Quicksort

Funkcja `main()`

Wiersze nr 6, 7 i 8 zawierają instrukcję wywołania `qsort()`. Jako pierwszy argument jest przekazywany do niej wskaźnik na tablicę do uporządkowania. Jest on rzutowany na typu `void *`. Następnie przekazywana jest liczba elementów tej tablicy obliczona za pomocą wyrażenia, które jest użyte także w funkcji `fill_array()`. Jako trzeci argument przekazywany jest rozmiar pierwszego elementu tej tablicy. Może to być rozmiar dowolnego elementu - wszystkie są takie same, język C gwarantuje, że pierwszy element tablicy zawsze będzie istniał. Jako ostatni argument wywołania przekazywany jest wskaźnik (nazwa) funkcji do porównywania wartości elementów tablicy.

Kopiec

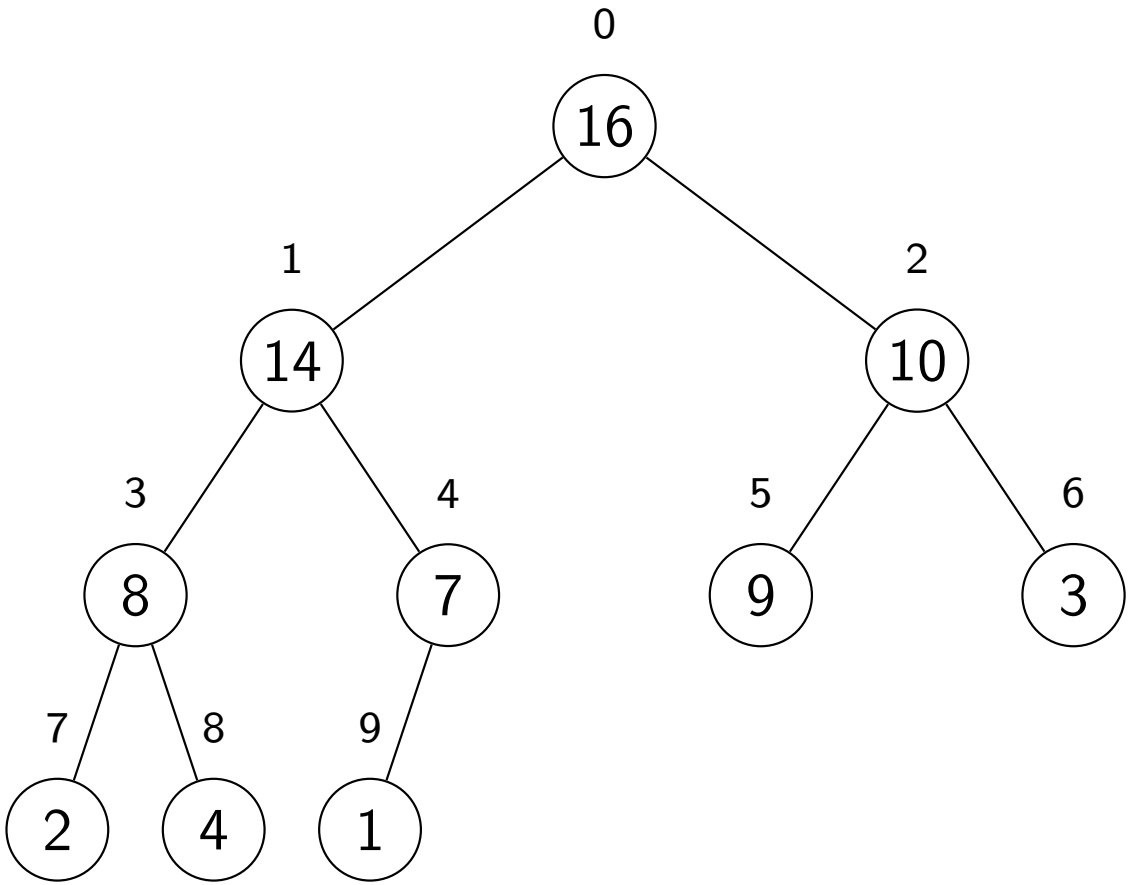
Kopiec, nazywany w polskiej literaturze również *stogiem*, jest drzewem binarnym, które ma kształt drzewa binarnego pełnego lub zupełnego i w którym zachodzi *własność kopca*[1]. Jeśli kopiec jest drzewem binarnym zupełnym, to braki w węzłach ostatniego poziomu mogą występować tylko po jego prawej stronie. Kopiec nie jest najczęściej realizowany w postaci dynamicznej struktury danych, a odwzorowywany w tablicy, w ten sposób, że jako klucz węzła przyjmowany jest indeks elementu w tablicy, a jako wartość tego węzła wartość elementu. Korzeń jest zawsze odwzorowywany na pierwszy element tej tablicy. Jeśli przyjmiemy, że ta tablica jest indeksowana od 1 i że *index* oznacza indeks w tablicy węzła wewnętrznego kopca, to indeks jego lewego potomka uzyskamy za pomocą wyrażenia $2 \cdot index$, prawego za pomocą $2 \cdot index + 1$, a przodka za pomocą $index/2$, gdzie $/$ oznacza dzielenie całkowite.

[1] W literaturze angielskiej kopiec jest określany słowem *heap*, które z kolei w polskiej terminologii informatycznej oznacza stertę. Aby nie powodować błędów interpretacji w naszej literaturze wprowadzono odrębny termin.

Kopiec

W języku C tablice są indeksowane od 0. Jeśli podstawimy za *index* choćby tę wartość, to przekonamy się, że podane na poprzednim slajdzie wzory przestaną funkcjonować dla takiej tablicy. Istnieją dwa wyjścia z tej sytuacji: można pominąć pierwszy element tablicy lub przekształcić odpowiednio wzory. My wybierzemy ten drugi wariant. Zatem wyrażenie do obliczania indeksu przodka będzie miało postać $(\mathit{index}+1)/2$. Indeks lewego potomka wyliczymy zaś korzystając ze wzoru: $2 \cdot \mathit{index} + 1$, a prawego z wyrażenia: $2 \cdot \mathit{index} + 2$. Następny slajd zawiera ilustrację kopca i jego odwzorowania w tablicy indeksowanej od zera.

Kopiec



0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Kopiec

Przedstawiony na poprzednim slajdzie kopiec jest *kopcem maksimum* (ang. *max-heap*), tzn. dla niego własność kopca jest określona następująco: $A[parent(i)] \geqslant A[i]$, czyli wartość przodka dowolnego węzła jest zawsze większa lub równa wartości tego węzła. A oznacza tablicę. Istnieją także *kopce minimum* (ang. *min-heap*). Są to kopce, których własność jest następująca: $A[parent(i)] \leqslant A[i]$. W dalszej części wykładu będziemy posługiwać się kopcem maksimum, który będziemy nazywać po prostu kopcem. Relacja między kopcem, a tablicą w której jest on odwzorowany jest określona nierównością *rozmiar kopca* $\leqslant$ *rozmiar tablicy*, gdzie przez *rozmiar* rozumiemy liczbę elementów tablicy lub liczbę węzłów kopca. Z tej nierówności wynika, że nie wszystkie elementy tablicy muszą należeć do kopca.

Heapsort

Kopce mogą być zastosowane do tworzenia tzw. kolejek priorytetowych (ang. *priority queue*), ale nas będzie interesować na tym wykładzie zastosowanie ich w algorytmie sortowania tablicy, który jest spokrewniony z algorytmem sortowania przez wybór, a nazywa się *Heapsort*. Ten algorytm, podobnie jak Quicksort realizuje sortowanie niestabilne. W porównaniu do tego ostatniego jest wolniejszy, choć nadal należy do najszybszych algorytmów tego typu, za to jego złożoność obliczeniowa wynosi $O(n \cdot log_2(n))$, dla wszystkich przypadków. Może on być zrealizowany zarówno w postaci rekurencyjnej (ta zostanie przedstawiona na wykładzie), jak i iteracyjnej.
Na kolejnych slajdach przedstawione są kody źródłowe funkcji obliczających indeksy prawego i lewego potomka węzła (wyliczanie indeksu przodka w tym algorytmie nie jest stosowane), a następnie funkcje, które przywracają własność kopca, budują kopiec i w końcu sortują tablicę.

Heapsort

Funkcja `get_left_child_index()`

```
1  static inline int get_left_child_index(int index)
2  {
3      return (index << 1) + 1;
4  }
```

Heapsort

Funkcja `get_left_child_index()`

Przedstawiona na poprzednim slajdzie funkcja wyznacza indeks lewego potomka węzła kopca. Aby przyspieszyć jej dzianie zamiast operatora mnożenia został zastosowany operator przesunięcia bitowego w lewo. Możemy tak postąpić, bo drugim argumentem mnożenia jest 2. Dodatkowo jest to funkcja typu *inline*, czyli jest rozwijana podobnie jak makro, ale przez kompilator (o ile on na to pozwala), a nie preprocesor.

Heapsort

Funkcja `get_right_child_index()`

```
1  static inline int get_right_child_index(int index)
2  {
3      return (index << 1) + 2;
4  }
```

Heapsort

Funkcja `get_right_child_index()`

Funkcja z poprzedniego slajdu wyznacza indeks prawego potomka węzła. Została ona napisana przy zastosowaniu podobnych technik, jak jej odpowiedniczka dla lewego węzła.

Heapsort

Funkcja heapify()

```
1  void heapify(int_array_type array, int index, unsigned int size)
2  {
3      int left = get_left_child_index(index),
4          right = get_right_child_index(index),
5          largest = index;
6      if(left<=size)
7          if(array[left]>array[index])
8                  largest = left;
9      if(right<=size)
10         if(array[right]>array[largest])
11             largest = right;
12     if(largest!=index) {
13         swap(&array[index],&array[largest]);
14         heapify(array,largest,size);
15     }
16 }
```

Heapsort

Funkcja `heapify()`

Funkcja `heapify()` jest podstawowym podprogramem w implementacji algorytmu Heapsort. Jej zadaniem jest przywrócenie własności kopca. Nie zwraca ona żadnych wartości, ale posiada trzy parametry. Przez pierwszy jest przekazywana do niej tablica z odwzorowanym kopcem, w którym jest zaburzona wspomniana własność. Przez drugi parametr przekazywany jest indeks elementu, którego wartość potencjalnie zaburza tę własność. Przez trzeci parametr przekazywany jest rozmiar kopca. Wewnątrz funkcji najpierw są wyznaczane i zapamiętywane w zmiennych lokalnych `left` i `right` indeksy lewego i prawego potomka węzła, co do którego zachodzi podejrzenie, że zaburza własność kopca (wiersze nr 3 i 4), oraz w zmiennej lokalnej `largest` zapamiętywana jest wartość indeksu tego węzła (wiersz nr 5). Ta zmienna będzie przechowywała indeks tego węzła spośród wspomnianej trójki, który ma największą wartość. Zatem w funkcji wstępnie przyjmujemy, że własność kopca nie jest zaburzona.

Heapsort

Funkcja `heapify()`

Następnie funkcja sprawdza, czy lewy potomek węzła istnieje, tzn. czy indeks zapamiętany w `left` związany jest z elementem, który mieści się w kopcu (wiersz nr 6) i jeśli tak jest, to sprawdza, czy wartość tego potomka jest większa od wartości przodka (wiersz nr 7). Jeśli i ten warunek okaże się prawdziwy, to w zmiennej `largest` jest zapamiętywany jego indeks. W wierszu nr 8 w podobny sposób funkcja sprawdza, czy istnieje prawy potomek węzła. Jeśli tak jest, to funkcja sprawdza, czy jego wartość jest większa od tego węzła, którego bieżąco zmienna `largest` określa jako największy (o największej wartości). Jest to na tym etapie albo węzeł o indeksie `index`, albo jego lewy potomek. Jeśli wartość prawego potomka jest większa od tego węzła, to indeks tego potomka jest zapamiętywany w zmiennej `largest` (wiersz nr 11). Zatem po wykonaniu wiersza nr 11 we wspomnianej zmiennej znajduje się indeks węzła o największej wartości spośród trójki: węzeł potencjalnie zaburzający własność kopca, jego lewy i prawy potomek.

Heapsort

Funkcja `heapify()`

W wierszu nr 12 funkcja sprawdza, czy wartość zmiennej `largest` jest różna od wartości parametru `index`. Jeśli tak by nie było, to oznaczałoby, że własność kopca nie była zakłócona i funkcja może zakończyć działanie, nie wykonując żadnych dodatkowych czynności. Jeśli ten warunek jest jednak prawdziwy, to należy zamienić miejscami wartości w węzłach, których indeksy określają zmienne `index` i `largest` (wiersz nr 13). To może jednak spowodować przeniesienie zaburzenia własności w dół kopca. Teraz może ona nie zachodzić w poddrzewie, w którym węzeł o indeksie `largest` jest korzeniem. Dlatego funkcja `heapify()` wywołuje się rekurencyjnie dla tego węzła (wiersz nr 14).

Heapsort

Funkcja build_heap()

```
1  void build_heap(int_array_type array)
2  {
3      int i;
4      const int number_of_elements =
5          sizeof(int_array_type)/sizeof(*array);
6      for(i=number_of_elements/2;i>=0;i--)
7          heapify(array,i,number_of_elements-1);
8  }
```

Heapsort

Funkcja `build_heap()`

Funkcja `build_heap()` buduje kopiec w tablicy, która ma być posortowana. Używa w tym celu opisanej wcześniej funkcji `heapify()`. Nie zwraca ona żadnej wartości, a przez parametr przyjmuje tablicę, w której utworzy kopiec. W wierszach nr 4 i 5 zdefiniowana jest w tej funkcji stała, która określa liczbę elementów tablicy. Kopiec jest budowany w pętli `for`. Proszę zwrócić uwagę, że ta funkcja iteruje po elementach tablicy rozpoczynając od środka tablicy i „przemieszczając się" ku elementowi pierwszemu. Powstaje pytanie, dlaczego nie jest porządkowana druga połowa tablicy. Otóż funkcja przyjmuje, że rozmiar kopca jest równy rozmiarowi tablicy. Zatem elementy należące do tej połowy są liśćmi kopca (lub, jak kto woli, tworzą jednoelementowe kopce). Stąd wywołując `heapify()` dla elementów z pierwszej połowy `build_heap()` zapewnia, że elementy w drugiej połowie też będą włączone do kopca - zadba o to `heapify()`.

Heapsort

Funkcja heapsort()

```
1  void heapsort(int_array_type array)
2  {
3      int last_index = sizeof(int_array_type)/sizeof(*array)-1;
4      int i;
5
6      build_heap(array);
7      for(i=last_index;i>0;i--){
8          swap(&array[0],&array[i]);
9          heapify(array,0,--last_index);
10     }
11 }
```

Heapsort

Funkcja `heapsort()`

Funkcja `heapsort()` dokonuje właściwego sortowania tablicy. Nie zwraca ona żadnej wartości, ale przez parametr przyjmuje tablicę do posortowania. Wewnątrz tej funkcji jest zadeklarowana i zainicjowana zmienna `last_index`, która zawiera indeks ostatniego elementu tablicy należącego do kopca i wyznacza pośrednio rozmiar tej struktury. Funkcja najpierw buduje kopiec w tablicy poprzez wywołanie `build_heap()`, a następnie w pętli `for` iteruje po całej tablicy, poczynając od elementu ostatniego (początkowo jest to też ostatni element kopca) do elementu pierwszego. W czasie tych iteracji zamienia ona miejscami wartość pierwszego elementu z tym, który jest określany przez licznik pętli (zmienna `i`), a następnie przywraca własność kopca poczynając od pierwszego elementu tablicy. Jakie jest uzasadnienie dla tych czynności? Otóż, w kopcu największa wartość znajduje się w pierwszym elemencie tablicy, a w posortowanej niemalejąco tablicy powinna się ona znaleźć w ostatnim. Należy zatem wartości tych elementów zamienić miejscami (wiersz nr 8). Ta zamiana może jednak zaburzyć własność kopca.

Heapsort

Funkcja `heapsort()`

Aby ją przywrócić funkcja `heapsort()` wywołuje `heapify()` dla pierwszego elementu tablicy. Tym razem jednak z kopca wyłączany jest ostatni element tablicy, bo on jest już uporządkowany. W każdej kolejnej iteracji pętli `for` wielkość kopca jest zmniejszana o jeden i zamieniane są miejscami wartości w pierwszym elemencie tablicy (korzeniu) i elemencie kopca o indeksie `i`. Po zakończeniu tej pętli cała tablica jest uporządkowana.

Heapsort

Funkcja `main()`

```
1  int main(void)
2  {
3      int_array_type array;
4      fill_array(array);
5      print_array(array);
6      heapsort(array);
7      print_array(array);
8      return 0;
9  }
```

Heapsort

Funkcja `main()`

Jedyna różnica w funkcji `main()` między przykładowymi programami dla Heapsort i Quicksort jest taka, że zamiast wywołania funkcji `quicksort()` wywoływana jest `heapsort()` (wiersz nr 6). Jako argument jej wywołania przekazywana jest tablica do uporządkowania.

Podsumowanie

Oba przedstawione algorytmy sortowania tablic należą do najszybszych w tej kategorii, ale w przypadku optymistycznym i średnim przewagę wykazuje Quicksort. Przypadkiem pesymistycznym dla tego algorytmu, dla którego jego złożoność obliczeniowa wynosi $\Theta(n^2)$ jest sortowanie tablicy już posortowanej. Wówczas algorytm dzieli obszary tablicy na dwa, z których jeden jest jednoelementowy, a w drugim znajdują się pozostałe elementy pierwotnego obszaru. Najlepsze dla tego algorytmu są podziały, w których nowe obszary otrzymują po połowie elementów obszaru wyjściowego. Aby uniknąć przypadku pesymistycznego stosowane są różne techniki. Można przez rozpoczęciem wykonania tego algorytmu zamienić miejscami wartości między kilkoma losowo wybranymi elementami tablicy. Nie ma wprawdzie gwarancji, że to działanie nie uporządkuje tablicy, ale jest duże prawdopodobieństwo, że zaburzy porządek wartości, jeśli tablica była już uporządkowana. Inny sposób polega na zamianie tego algorytmu w algorytm probabilistyczny, który wybiera wartość osiującą (pseudo)losowo. Żadna z tych technik nie daje jednak całkowitej pewności, że przypadek pesymistyczny nie wystąpi.

Podsumowanie

Algorytm Heapsort ma również tę przewagę nad algorytmem Quicksort, że da się wyeliminować z jego implementacji rekurencję, zatem nie będzie ona tworzyła narzutu związanego z użyciem pamięci na stosie. Złożoność przestrzenna tego algorytmu może zatem być stała[2].
W praktyce algorytm Quicksort wydaje się być częściej używany niż Heapsort. Jednak w niektórych zastosowaniach bezpieczniej jest użyć tego drugiego. Przykładem mogą być usługi zdalne, które używają sortowania. Jeśli użyty byłby w nich algorytm Quicksort to mogłyby być one narażone na ataki typu Denial of Service (DoS) za pomocą żądań ich wykonania z odpowiednio spreparowanymi danymi wejściowymi.

[2] Takie implementacje algorytmu Heapsort zostały opisane m.in. w książkach „Perełki oprogramowania" Jona Bentley'a oraz „Algorytmy i struktury danych" A. V. Aho, J. E. Hopcrofta i J. D. Ullmana.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!