Sortowanie kopcowe

Andrzej Salwicki

4 stycznia 2015

Streszczenie

W tym artykule analizujemy program sortowania kopcowego zbudowany z wykorzystaniem dziedziczenia.

1 Wstęp

Antoni Kreczmar zaprogramował w Loglanie algorytm heapsort wykorzystując możliwość dziedziczenia klasy przez procedurę.

Zacznijmy od przypomnienia czym jest kopiec.

Definicja 1. *Kopiec jest to drzewo binarne wywazone, tzn. takie że dla pewnej liczby naturalnej k, kazdy liść tego drzewa jest na poziomie k lub k-1 i ponadto dla każdego węzła n tego drzewa wartość wpisana w tym węźle jest nie mniejsza niż wartości znajdujące sie w poddrzewie węzła n.*

W naszym przypadku kopiec bedzie zapisany w tablicy.

Lemat 1. *Niech A bedzie tablicą n-elementową* $A[1], A[2], \ldots, A[n]$*.Jeżeli dla kazdego k,* $1 \leq k \leq n$ *zachodzi* $A[k] \leq A[kdiv2] \leq A[kdiv4] \leq \cdots \leq A[1]$ *to tablica A reprezentuje kopiec.*

Dowód. Łatwo zauważyć, ze tablica A może być pojmowana jako drzewo binarne. Korzeniem drzewa jest element $A[1]$. Dla każdego $i, 1 \leq i \leq n$, element $A[i]$ jest ojcem synów: lewego $A[2i]$ oraz prawego $A[2i+1]$ pod warunkiem, że elementy te istnieja w tablicy A tzn. że $2i \leq n$ i odpowiednio $2i+1 \leq n$. Nietrudno zauważyć, że drzewo jest wyważone. Warunek wymieniony w tezie lematu gwarantuje, że dla każdego węzła $A[l]$ wszystkie węzły poddrzewa $A[l]$ zawierają elementy nie mniejsze od $A[l]$. □

2 Konstruowanie algorytmu

Zacznijmy od analizy następującego kodu

Kopcuj:

```
i :=l; j := 2*i; x :=A(i);
while j<=p do
   if j<p and A(j)<A(j+1)
   then
     j:=j+1
   fi;
   if x>=A(j) then exit fi;
   A(i):=A(j); i:=j; j:=2*i;
done;
A(i) := x;
```

Niech dane będą warunki: warunek wstępny
α: *tablica A na miejscach* $l+1, \ldots, p$ *reprezentuje las kopców,*
oraz warunek końcowy
β: *tablica A na miejscach* $l, l+1, \ldots, p$ *reprezentuje las kopców.*

Lemat 2. *Program Kopcuj jest poprawny ze względu na warunek początkowy* α *i warunek końcowy* β
$\alpha \implies$ [Kopcuj] β.

Dowód. Gdy $j > p$ to $A[l]$ jest liściem, $A[l]$ jest też korzeniem. A więc fragment tablicy $A[l], A[l+1], \ldots, A[p]$ jest lasem kopców. Rozważmy przypadek gdy $j \leq p$. Zacznijmy od obserwacji, że po wykonaniu instrukcji
if j<p and_if A[j] < A[j+1] then j := j+1 fi
zachodzi $A[j] = max\{A[2i], A[2i+1]\}$. W przypadku gdy x >A[j] to z założenia indukcyjnego x jest większe od wszystkich węzłów poddrzewa A[j], a więc i od wszystkich węzłow poddrzewa A[j-1]. Nie ma nic więcej do roboty i opuszczamy (exit) petlę. Jeśli tak nie jest to kładąc A[i]:=A[j]; i:= j; j:=2*i; sprowadzamy nasz problem do problemu przesiewania w poddrzewie drzewa początkowego. Po pewnej liczbie powtórzeń algorytm zatrzyma się ponieważ kolejne drzewo będzie liściem. □

Rozpatrzmy teraz następujący kod

BS:

```
l:= (upper(A) div 2)+1;
do
   l:= l-1;
   if l=lower(A) then exit fi;
   [Kopcuj]
od
```

Lemat 3. *Zakładam, że lower(A)=1.Program BS buduje kopiec z elementów tablicy A.*

Dowód. Po wykonaniu instrukcji l:=(upper(A)div 2)+1 fragment tablicy A: A[l], A[l+1], ...,A[upper(A)] jest lasem drzew. W każdej iteracji pętli do ...od powiększamy fragment zawierający las kopców. Powtarzanie zakończy się gdy l=lower(A). Wtedy las kopców jest jednym kopcem o korzeniu A[1]. □

Teraz rozważmy kod

ST:
```
p:= upper(A); l:=lower(A);
do
   x:=A[l]; A[l]:=A[p]; A[p]:= x;
   p:=p-1;
   if p=lower(A) then exit fi;
   Kopcuj
od
```

Lemat 4. *Jeśli tablica A jest kopcem, to program ST sortuje elementy tablicy A w porządku niemalejącym.*

Dowód. W pierwszym kroku iteracji zamieniamy miejscami A[l] i A[p], ponadto zmniejszamy wartość p. Ostatnim elementem tablicy A jest więc element największy. Gdy l=p to tablica A jest uporządkowana niemalejąco i można zakończyć działanie programu ST. W przeciwnym przypadku może być tak, że fragment A[l], ..., A[p] tablicy A nie jest kopcem. Ale po wykonaniu polecenia Kopcuj fragment ten będzie kopcem. Kolejny co do wielkości element trafi na miejsce p w tablicy A. Proces ten kończy się i z chwilą jego zakończenia tablica A jest uporządkowana niemalejąco. □

Co dalej? Jak skonstruować procedurę heapsort?
Można zaproponować trzy rozwiązania:

- Treścią procedury mogą stać się programy BS i ST, w których powtarza się program Kopcuj.
- Treść procedury może polegać na wywołaniu po kolei procedur BudujKopiec i Sortuj, każda z tych procedur wywołuje procedurę Kopcuj.
- Na treść procedury heapsort składają się klasa Przesiej, procedura BudujKopiec rozszerzająca klasę Przesiej i procedura Sortuj też rozszerzająca klasę Przesiej.

Zestawmy te trzy możliwości obok siebie

```
unit heapsortA: procedure
    <deklaracje>
begin
    BS;
    ST
end heapsortA
```

```
unit heapsortB: procedure
    unit Kopcuj: procedure
    end Kopcuj
    unit BudKop: procedure
    ... call Kopcuj ...
    end BudKop
    unit Sortuj: procedure
    ... call Kopcuj ...
    end Sortuj
begin
    call BudKop;
    call Sortuj;
end heapsortB
```

```
unit heapsortC: procedure
    unit Przesiej:
      class ...
    unit BudKop:
      Przesiej procedure
    unit Sortuj:
      Przesiej procedure
begin
    call BudKop;
    call Sortuj;
end heapsortC
```

Każda z tych propozycji pozwoli skonstruować poprawną procedurę heapsort. Która droga prowadzi do najlepszego rozwiązania? Procedura heapsortB ma największy koszt. Podczas wykonywania instrukcji BudKop i Sortuj instrukcja call Kopcuj będzie wykonywana wielokrotnie. Wielokrotnie trzeba będzie tworzyć rekord aktywacji tej procedury. Procedura heapsortC pozwala uniknąć tego narzutu. Zobaczymy jak. Ale najbardziej wydajna jest chyba procedura heapsortA. Tyle, że wymaga to od nas dwukrotnego napisania kodu Kopcuj, wewnatrz programu BS i wewnątrz programu ST. Nie ma w tym nic złego tak długo jak długo nie trzeba wprowadzać zmian w kodzie Kopcuj – musimy pamietać, że trzeba to zrobić w dwu miejscach!

Dokończmy więc konstrukcję procedury heapsortC. Tworzymy klasę Przesiewanie.

```
unit Przesiewanie: class;
    var koniec: boolean
begin
    do
        inner ;
        if koniec then exit fi;
        i :=l; j := 2*i; x :=A(i);
        while j<=p do
            if j<p and A(j)<A(j+1)
            then
                j:=j+1
            fi;
            if x>=A(j) then exit fi;
            A(i):=A(j); i:=j; j:=2*i;
        done;
        A(i) := x;
    done
end przesiewanie;
```

i procedury dziedziczące klasę Przesiewanie: budujKopiec i sortuj.

```
unit budujKopiec: Przesiewanie procedure;
begin
        koniec := (l=lower(A)); l :=l-1;
end budujKopiec;
```

Nie trzeba długo dowodzić, że prawdziwą treść procedury budujKopiec stanowi treść klasy Przesiewanie w której pseudoinstrukcję inner zastąpiono dwoma instrukcjami z deklaracji procedury budujKopiec. czyli należy pamiętać, że dzieki regule konkatenacji powinniśmy przyjąć, że deklaracja procedury budujKopiec to

```
unit budujKopiec: procedure;
    var koniec: boolean
begin
    do
        koniec := (l=lower(A)); l :=l-1;
        if koniec then exit fi;
        i :=l; j := 2*i; x :=A(i);
        while j<=p do
            if j<p and A(j)<A(j+1)
            then
                j:=j+1
            fi;
            if x>=A(j) then exit fi;
            A(i):=A(j); i:=j; j:=2*i;
        done;
        A(i) := x;
    done
end budujKopiec;
```

Wcześniej przekonaliśmy się, że ten algorytm przekształca tablicę A w kopiec. Podobnie można sprawdzić, że instrukcja call sortuj uporządkuje tablicę A w ciąg niemalejacy.

```
unit sortuj: Przesiewanie procedure;
begin
    d:= lower(A); koniec:=(p=d);
    x:=A(d); A(d) := A(p); A(p) := x;
    p := p-1;
end sortuj;
```

Po przyjęciu tych trzech deklaracji łatwo uzupełnić treść procedury heapsortC tak jak to widać poniżej.

```
unit heapsortC: procedure(A: arrayof integer);
    var i, j, l, p, x: integer;
    unit Przesiewanie: class ...
    unit budujKopiec: Przesiewanie procedure ...
    unit sortuj: Przesiewanie procedure ...
begin
    l := upper(A) div 2 +1;
    p := upper(A);
    call budujKopiec;
    call sortuj
end heapsortC
```

3 Zadania

1. Napisz kompletny program, który wczytuje zadany ciąg liczb i sortuje go.

2. Napisz program zawierający procedury heapsortB i heapsortC. Tworzy (dość dużą) tablicę liczb, np. wybierając liczby pseudolosowe i porządkuje ją dwukrotnie, raz przy pomocy heapsortB i drugi raz przy pomocy heapsortC. Porównaj czasy działania tych algorytmów.

3. Czy potrafisz zmienić procedurę heapsort tak by porządkowała tablice elementów typu T?

4 Algorytm

```
unit heapsort: procedure(A: arrayof integer);

    var i,j,l,d,p,x: integer

    unit Przesiewanie: class;
        var koniec: boolean
    begin
        do
            inner ;
            if koniec then exit fi;
            i :=l; j := 2*i; x :=A(i);
            while j<=p do
                if j<p and A(j)<A(j+1)
                then
                    j:=j+1
                fi;
                if x>=A(j) then exit fi;
                A(i):=A(j); i:=j; j:=2*i;
            done;
            A(i) := x;
        done
    end Przesiewanie;

    unit budujKopiec: Przesiewanie procedure;
    begin
        koniec := (l=lower(A)); l :=l-1;
    end budujKopiec;

    unit sortuj: Przesiewanie procedure;
    begin
        d:= lower(A); koniec:=(p=d);
        x:=A(d); A(d) := A(p); A(p) := x;
        p := p-1;
    end sortuj;

(* tu zaczynają się instrukcje procedury heapsort*)
begin
    l:= upper(A) div 2 +1;
    p:= upper(A);
    call budujKopiec;
    call sortuj;
end heapsort;
```