Съдържание

1 Сортировки **2**

1.1 Наивни сортировки 2

1.2 MergeSort 2

1.3 HeapSort 2

1.3.1 Приоритетна опашка, пирамида. 2

1.3.2 Алгоритъм HeapSort 7

1.4 QuickSort 9

1.4.1 Предимства и недостатъци на QuickSort 12

1.4.2 Подобрения на QuickSort 12

Използвана литература **16**

Глава 1

Сортировки

Тук трябва да дефинираме задачата сортиране, обозначенията и какво е инверсия

1.1 Наивни сортировки

1.2 MergeSort

1.3 HeapSort

1.3.1 Приоритетна опашка, пирамида.

Приоритетна опашка ще наричаме абстрактна структура данни, която съхранява множество от обекти S, всеки елемент на S е двойка от вида $\langle key, value \rangle$. Върху приоритетната опашка са дефинирани поне две операции – $Insert(S, el)$, която добавя нов елемент в структурата и функцията $ExtractMax(S)$, която връща елемента с най-голям ключ key и го изтрива от структурата.

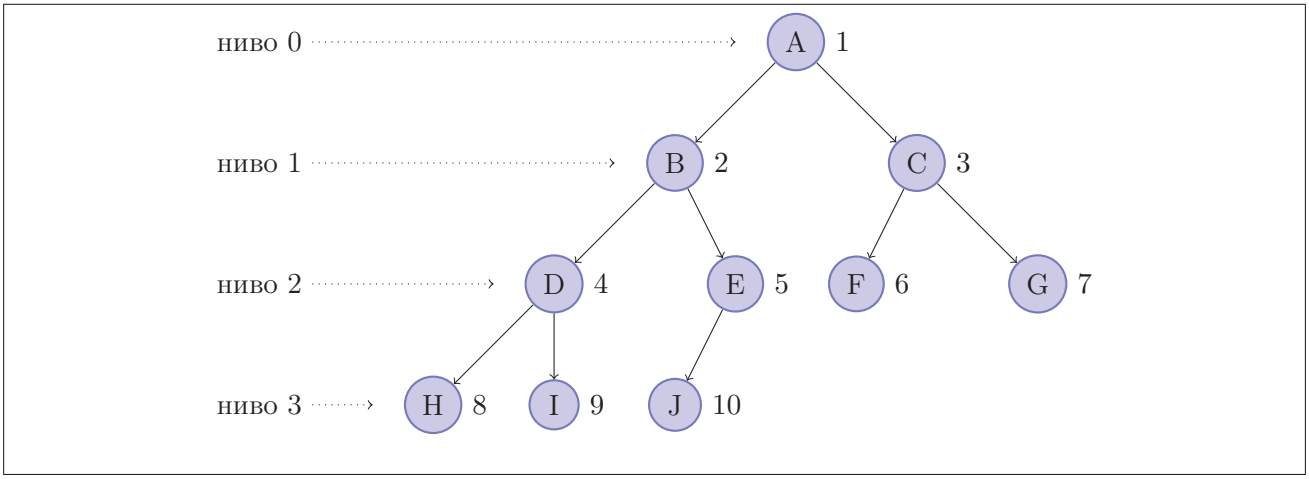
Вероятно тази структура е била дефинирана за първите операционни системи, които са управлявали реда на изпълнението на група процеси (jobs) върху споделен компютър с времеделене. Такъв компютър изпълнява в даден момент един процес (задание), но много потребители подават своите задания за изпълнение, като всяко задание има приоритет *key*. Когато текущият процес завърши или бъде прекъснат, планиращата програма на системата *job scheduler* извлича заданието с най-голям приоритет от опашката и го стартира. Ако текущият процес е бил прекъснат, той се вмъква в опашката, за да чака последващо включване. Когато в системата постъпи ново задание, то се вкарва в приоритетната опашка.

Има няколко наивни реализации на приоритетната опашка:

- Ако съхраняваме елементите от S в обикновен масив или списък, процедурата $Insert(S, el)$ ще се изпълнява за време $O(1)$, а $ExtractMax(S)$ – за време $O(n)$.
- Ако съхраняваме елементите от S в сортиран списък, процедурата $Insert(S, el)$ ще се изпълнява за време $O(n)$, а $ExtractMax(S)$ – за време $O(1)$.

При наивните реализации поне една от двете основни операциите се изпълнява бавно – за време $O(n)$.

По-долу ще опишем най-леката и най-използвана ефективна реализация на приоритетна опашка: тя се нарича двоична пирамида (binary heap). При нея двете основни операции отнемат време $O(\lg(n))$.



Фигура 1.1: Попълнено двоично дърво: всички нива са запълнени с изключение на последното, където са запълнени няколко върха плътно в лявата част на нивото.

Двоично дърво ще наричаме кореново дърво, всеки връх v на което има най-много два наследника, които ще наричаме ляв и десен син на v.

Попълнено двоично дърво ще наричаме двоично дърво, за което всички нива (ниво е множество от върхове, равноотдалечени от корена) са запълнени, с изключение на последното, което е частично запълнено, отляво надясно (виж Фигура 1.1).

Обикновено дърветата се представят в паметта на компютъра с помощта на указатели между върховете, които представят ориентираните ребра на дървото.

Попълненото двоично дърво може да се представи по-икономично – в обикновен масив, всеки връх се разполага в елемент на масива с номер, който се получава така: върховете се номерират от корена към следващите го нива последователно – както е показано на Фигура 1.1. Коренът винаги има номер 1, левият му син получава номер 2, десният – 3, върховете от 2-ро ниво получават номера от 4 до 7, върховете от ниво k получават номера от 2^k до $2^{k+1} - 1$.

Лесно се събразява, че при такава номерация са в сила следните зависимости между връх с номер i и свързаните с него: неговият родител има номер $\lfloor i/2 \rfloor$, левият му син има номер $2i$, а десният – $2i + 1$. По-нататък ще означаваме номерата на родините на връх с номер i съответно с $parent(i)$, $left(i)$ и $right(i)$. Тяхното изчисляване е лесно (ротация на битовете на i една позиция надясно или наляво с дописване на 0 или 1) и в практическа реализация следва да бъдат изчислявани пряко или дефинирани като макроси.

Очевидно е също, че попълнено дърво с n върха ще заеме елементите на масива с последователни номера от 1 до n. За удобство ще съкращаваме 'връх с номер i' до 'връх i'.

Поддървото, породено от върха i ще означаваме с $[[i]]$. Определяме го така:

- Ако върхът i е листо, $[[i]]$ съдържа единствен връх i.
- Ако върхът i има наследници (наследник), $[[i]]$ е кореново дърво с корен връх i, ляво поддърво $[[left(i)]]$ и дясно поддърво $[[right(i)]]$.

Всички породени поддърветата на попълнено дърво са попълнени дървета.

Попълнено дърво с n върха има точно $parent(n) = \lfloor n/2 \rfloor$ върхове с наследници. Това следва от факта, че функцията $parent(i)$ е монотонно растяща: върхът n ще има родител с най-голям номер. От равенството $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$ следва, че листата в дърво с n върха са $\lceil n/2 \rceil$.

Ще съхраняваме структурата S в масив $A[1 \dots n]$, заедно с брояч $size$, който ще отчита колко елемента в даден момент са вътре в приоритетната опашка. Очевидно в S можем да вкараме най-много n елемента. Елементите при практическа реализация са от вида $\langle key, value \rangle$, като имаме линейна наредба по key. За простота ще предполагаме, че key е цяло число и ще пропускаме $value$.

h-инверсия ще наричаме двойка индекси $\langle i, j \rangle, 0 < j < i \leq size$, такава, че j се намира на пътя от i нагоре към корена на попълненото дърво и $A[i] > A[j]$.

Ще казваме, че S е *пирамида*, когато за всяко $i, 1 < i \leq size$ е изпълнено неравенството $A[i] \leq A[parent(i)]$.

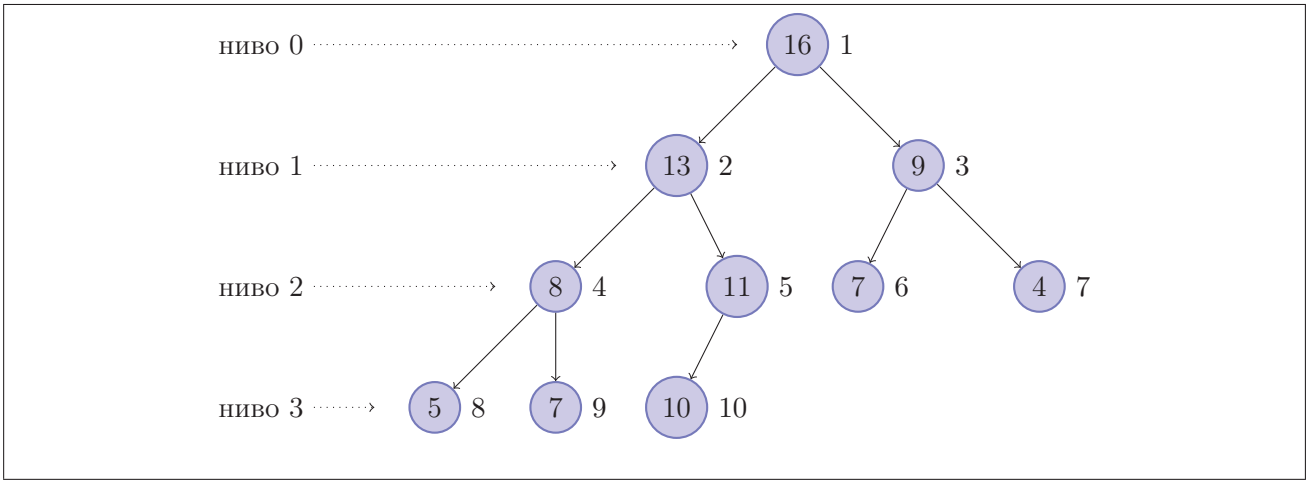
Лема 1.1. *S е пирамида $\iff$ В S няма h-инверсии.*

Доказателство:
$\Longrightarrow$ Всеки път от връх нагоре към корена в попълненото дърво на S посещава растяща редица стойности на ключове в A: ако $(v_1, v_2, \dots, v_l)$ е такъв път, за всеки два съседни върха в него $v_{i+1} = parent(v_i)$ и $A[v_i] \leq A[v_{i+1}]$, в такава редица няма h-инверсии.
$\Longleftarrow$ По тривиалния път $(i, parent(i))$ няма h-инверсия, следователно $A[i] \leq A[parent(i)]$. □

От лемата следва, че в корена на пирамидата стойността на ключа ще е максимална.

Фигура 1.2 представя пирамида с 10 елемента.

Масивът A съдържа елементите $(16, 13, 9, 8, 11, 7, 4, 5, 7, 10)$. Той не е сортиран, но пирамидалното свойство е изпълнено: всеки елемент е по-малък или равен на родителя си. Коренът съдържа максималния елемент.



Фигура 1.2: Пирамида. Във всеки връх е изписана стойността на ключа, а встрани е изписана позицията на върха в масива A.

Да предположим, че в пирамида сме променили стойността (на ключа) в един от върховете. Тогава е възможно пирамидалното свойство да се наруши. Наричаме този връх *дефектен*, а получената структура *пирамида с дефект*.

Формална дефиниция: Казваме, че S е пирамида с дефект i, когато i участва във всички h-инверсии на S.

Лема 1.2. *Нека S е пирамида с дефект i. Тогава е вярно едно от двете твърдения:*

1. *Всички h-инверсии в S са от вида $\langle i, j \rangle$, като j е връх по пътя нагоре от i към корена.*
2. *Всички h-инверсии в S са от вида $\langle j, i \rangle$, като j е наследник (син или пранаследник) на i (j е от поддървото $[[i]]$).*

Доказателство:
Да допуснем, че има h-инверсия на i с негов наследник j_1, и друга, с негов обобщен родител (родител или прародител) j_2. От тези две h-инверсии следват неравенствата $A[j_1] > A[i] > A[j_2]$. Пътят нагоре към корена от j_1 ще мине първо през i, а после и през j_2. Оказва се, че двойката $\langle j_1, j_2 \rangle$ образува h-инверсия, което е противоречие. □

Лемата дава ясна характеризация на дефект в пирамида:

В случай 1. дефектът е по-голям от някой обобщен родител. Ще наричаме този случай *дефект нагоре*. Следната проста процедура намаля броя на h-инверсиите при дефект нагоре във върха i:

```
StepUp(A, i)
  1  swap(A[i], A[parent(i)])
```

Коректност: По пътя към корена най-малък е прекият родител, т.е. върховете i и $j = parent(i)$ образуват h-инверсия и размяната на стойностите им унищожава тази h-инверсия.

В поддървото $[[j]]$ преди размяната няма други h-инверсии и размяната няма да създаде нови: дефектът е по-голям от всички елементи на $[[j]]$ и след размяната той се оказва на върха на поддървото, а стойността на $A[j]$ преди размяната е по-голяма от всички наследници в поддървото $[[i]]$ (защото не е в h-инверсия с тях) и след размяната тя се оказва на върха на това поддърво.

Следователно след размяната ще останат h-инверсии само между връх j (новото място на дефекта) и неговите обобщени родители (тези h-инверсии се запазват при размяната). □

За да унищожим всички h-инверсиите при *дефект нагоре* във върха i, трябва да местим циклично дефекта нагоре, докато изчезне:

```
MoveUp(A, i)
  1  while (i > 1) ∧ (A[i] > A[parent(i)]) do
  2        swap(A[i], A[parent(i)])
  3        i ← parent(i)
```

Коректността на $MoveUp$ следва от свойството на $StepUp$ да издига дефекта на едно ниво по-близо до корена. Тъй като броят на нивата в попълненото дърво не надвишава $\lg(n)$, сложността на $MoveUp$ е $O(\lg(n))$.

В случай 2. на лема 1.2 ще казваме, че е налице *дефект надолу*. Такъв дефект потъва едно ниво надолу при следната проста процедура (предполагаме, че дефекта е във връх i и той има двама сина):

```
StepDown(A, i)
  1  if  A[left(i)] < A[right(i)]
  2        swap(A[i], A[right(i)])
  3  else  swap(A[i], A[left(i)])
```

Коректност: Ще разгледаме само случая когато $A[left(i)] < A[right(i)]$, другият е симетричен.

$A[right(i)]$ е най-големият от всички наследници на i, защото в поддърветата на двата сина няма h-инверсии, следователно синовете са максимални в своите поддървета и проверката на ред 1 показва, че десният син е по-голям от левия.

Върховете i и $j = right(i)$ образуват h-инверсия (i е дефектен, а j е максимален) и размяната на стойностите им ще я унищожи.

Тъй като десният син при размяната отива в корена на поддървото $[[i]]$, всички инверсии на корена с левите му наследници ще изчезнат.

Всички h-инверсии между дефекта и наследниците на j ще се запазят, но дефектът слиза едно ниво надолу, на мястото на десния си син.

Следователно след размяната ще останат h-инверсии само между връх j (новото място на дефекта) и неговите наследници. □

За да унищожим всички h-инверсии при *дефект надолу* във върха i (ако има такива), трябва да местим циклично дефекта надолу, докато изчезне:

```
HEAPIFY(A, i)
 1  NoInversions ← FALSE
 2  repeat
 3      if (left(i) ≤ size) ∧ (A[i] < A[left(i)])
 4          largest ← left(i)
 5      else largest ← i
 6      if (right(i) ≤ size) ∧ (A[largest] < A[right(i)])
 7          largest ← right(i)
 8      if largest ≠ i
 9          swap(A[i], A[largest])
10          i ← largest
11      else NoInversions ← TRUE
12  until NoInversions
```

Коректност: Доказателството ще извършим по индукция спрямо дълбочината на поддървото $[[i]]$, при изискването всеки път когато изпълняваме ред 2, i да участва във всички h-инверсии в поддървото $[[i]]$ и стойността на промеливата $NoInversions$ да е $FALSE$.

Очевидно, когато i е листо (дълбочината на поддървото е 0, в него не може да има h-инверсии), алгоритъмът ще приключи работата си.

Нека дълбочината на поддървото $[[i]]$ е $k+1$. Алгоритъмът проверява дали i има синове и h-инверсии. Сравняването на стойностите във връх i и синовете е достатъчно, тъй като в поддърветата на синовете няма h-инверсии и те са максимални елементи в своите поддървета.

Условието $largest \neq i$ на ред 8 е изпълнено точно когато в $[[i]]$ има поне 1 h-инверсия, върхът $largest$ е по-големият син и размяната на стойностите на върховете i и $largest$ ще свали дефекта едно ниво надолу (виж разсъжденията за коректността на *StepDown*).

На ред 10 i приема стойност $largest$, това е новото място на дефекта (ако въобще има дефект), променливата $NoInversions$ запазва стойността си $FALSE$, а дълбочината на поддървото $[[i]]$ е вече k. Цикълът ще се изпълни отново, но съгласно индукционното предположение дефектът ще изчезне.

Условието $largest \neq i$ на ред 8 не е изпълнено точно когато в подървото $[[i]]$ няма h-инверсии. Тогава променливата $NoInversions$ приема стойност $TRUE$ и изпълнението на програмата ще завърши с унищожен дефект. □

Броят изпълнения на цикъла *repeat* е най-много дълбочината на поддървото $[[i]]$, тази дълбочина е $\lfloor \lg(n) - \lg(i) \rfloor$, за скоростта на $Heapify$ по-нататък ще ползваме фината оценка $O(\lg(n/i))$ или по-грубата $O(\lg(n))$.

Процедурата $Heapify$ може да се съкрати така (предложение на Григор Цонев):

```
HEAPIFY(A, i)
 1  largest ← i
 2  repeat
 3      i ← largest
```

```
 4       if (left(i) ≤ size) ∧ (A[i] < A[left(i)])
 5           largest ← left(i)
 6       if (right(i) ≤ size) ∧ (A[largest] < A[right(i)])
 7           largest ← right(i)
 8       if largest ≠ i
 9           swap(A[i], A[largest])
10   until i = largest
```

Оставям на читателя да докаже еквивалентността на двете реализации.

Сега вече можем да дефинираме функциите, които реализират приоритетна опашка чрез пирамида. Определихме по-горе структурата S като двойка от масива A с n елемента и променливата $size$, която съхранява броят елементи в опашката.

```
CreateHeap(A)
 1  size ← 0
```

```
Insert(A, el)
 1  if size < n
 2      size ← size + 1
 3      A[size] ← el
 4      MoveUp(A, size)
 5  else
 6      PRINT 'ERROR: heap overflow!'
```

```
ExtractMax(A)
 1  if size > 0
 2      el ← A[1]
 3      A[1] ← A[size]
 4      size ← size − 1
 5      if size > 0
 6          Heapify(A, 1)
 7      return el
 8  else
 9      PRINT 'ERROR: heap is epmty!'
```

1.3.2 Алгоритъм HeapSort

Алгоритъмът HeapSort, както и реализацията на приоритетна опашка чрез пирамида са предложени от J. W. J. Williams през 1964 г.

Идеята на алгоритъма е проста – първо пъхаме всички елементи на масива в приоритетна опашка, после ги изваждаме един по един. При изваждането най-напред ще получим най-големия елемент, после втория по големина и т.н., тоест поредицата получена при изваждането ще е сортирана.

Можем да реализираме и двете фази на алгоритъма да работят на място – част от обработвания масив ще е двоична пирамида, а останалата част ще съдържа необработените елементи през фазата на изграждане на пирамидата или вече сортираните елементи през фазата на изваждането на елементите от пирамидата.

Фазата на изграждане се реализира с процедурата $MoveUp$, определена в предната секция:

BuildHeap0($A[1 \dots n]$)
1 $size \leftarrow 0$
2 **while** $size < n$ **do**
3 $size \leftarrow size + 1$
4 $MoveUp(A, size)$

Коректността следва от верността на следната проста ивариантa: При всяко изпълнение на ред 2 масивът $A[1 \dots size]$ е пирамида.

Сложността на $BuildHeap0$ е $\Theta(n \lg n)$.

През същата 1964 г. Robert W. Floyd предлага алгоритъм за изграждането на пирамида с линейна сложност:

BuildHeap1($A[1 \dots n]$)
1 $size \leftarrow n$
2 **for** $i \leftarrow parent(n)$ **downto** 1
3 $Heapify(A, i)$

Коректност: Ще докажем верността на инвариантното свойство: При всяко достигане на ред 2 поддърветата $[[k]]$, породени от върхове с номера $i < k \leq n$ са пирамиди (не съдържат h-инверсии).

При първото достигане на цикъла $i = parent(n)$. Функцията *parent* е монотонно растяща, следователно върховете k, за които $i < k$ са листа в попълненото двоично дърво, записано в масива A (i е върхът с най-голям номер, който има наследници). След като са листа, поддърветата $[[k]]$ са и пирамиди.

Когато се изпълнява тялото на цикъла за някакво i, неговите наследници имат по-големи номера и от верността на инвариантата следва, че поддърветатта им са пирамиди. Поддървото $[[i]]$ ще е пирамида с дефект i, на ред 3 този дефект ще бъде поправен от процедурата $Heapify$. Така ще се гарантира верността на инвариантата след намалянето на i с единица и следващото достигане на началото на цикъла.

При последното преминаване през ред 3 променливата i ще стане 0, програмата ще завърши, от верността на инвариантата следва, че за $0 < k$ поддървото $[[k]]$ е пирамида, но за $k = 1$ това поддърво е целият масив A. □

Сложност: Тривиална горна граница за сложността $O(n \lg n)$ следва от сложността на $Heapify$, изчислена в предната секция.

Можем да направим по-фина оценка, като отчетем че $Heapify$ стартирана върху елемент i има сложност $O(k)$, където k е дълбочината (броим и корена) на поддървото $[[i]]$.

При анализа на коректността по-горе показахме, че от монотонното нарастване на функцията *parent* следва, че върховете с номера $1 \dots parent(n)$ имат наследници, а върховете-листа са $parent(n) \dots n$. С други думи броят на поддърветата в пирамидата с дълбочина поне 2 е $parent(n) = \lfloor \frac{n}{2} \rfloor$, а тия с дълбочина 1 са $n - \lfloor \frac{n}{2} \rfloor$.

Пак от монотонността на *parent* следва, че върхът с най-голям номер и поддърво с дълбочина 3 ще е $parent(parent(n)) = \lfloor \frac{n}{4} \rfloor$, следователно поддърветата с дълбочина поне 3 са $\lfloor \frac{n}{4} \rfloor$, а тия с дълбочина 2 са точно $\lfloor \frac{n}{2} \rfloor - \lfloor \frac{n}{4} \rfloor$.

С индукция можем да докажем, че броят на поддърветата в попълненото дърво на пирамидата с дълбочина k ще е $\lfloor \frac{n}{2^{k-1}} \rfloor - \lfloor \frac{n}{2^k} \rfloor$.

Нека c е константата от асимптотичната оценка на $Heapify$, тоест времето за работа на $Heapify$ върху елемент i с дълбочина на поддървото k е не повече от ck.

Сумарното време за работата на $BuildHeap1$ по всички върхове с дълбочина k ще е ограничено

от израза $ck(\lfloor\frac{n}{2^{k-1}}\rfloor - \lfloor\frac{n}{2^k}\rfloor)$.

Сложността на $BuildHeap1$ е сума по k на израза по-горе:

$$T(n) \leq c\sum_{k=2}^{\lfloor \lg n \rfloor} k(\lfloor\frac{n}{2^{k-1}}\rfloor - \lfloor\frac{n}{2^k}\rfloor)$$

Можем да заместим лявата сума с по-голяма:

$$T(n) \leq c\sum_{k=1}^{\infty} k(\lfloor\frac{n}{2^{k-1}}\rfloor - \lfloor\frac{n}{2^k}\rfloor)$$

Горната сума е крайна (проверете сами!). Всеки член от вида $\lfloor\frac{n}{2^k}\rfloor)$ (освен първия, за $k = 0$) се среща два пъти – най-напред със знак минус и коефицент k, след това с положителен знак и коефицент $k + 1$. Събирайки двете срещания, ще получим всеки от членовете с коефицент 1 (включително и първия), така получаваме по-проста сума:

$$T(n) \leq c\sum_{k=0}^{\infty} \lfloor\frac{n}{2^k}\rfloor$$

Премахваме закръглението и получаваме по-голяма сума, която пресмятаме:

$$T(n) \leq c\sum_{k=0}^{\infty} \lfloor\frac{n}{2^k}\rfloor \leq c\sum_{k=0}^{\infty} \frac{n}{2^k} = cn\sum_{k=0}^{\infty} \frac{1}{2^k} = 2cn$$

□

Сложността на $BuildHeap0$ в средния случай също е линейна. Това следва от прости статистически съображения, а константата и е малка. С $BuildHeap$ по-долу ще бележим някоя от двете процедури за изграждане на пирамидата, тъй като е възможно в практически реализации да бъде предпочетена коя да е от тях.

Вече можем да опишем алгоритъма $HeapSort$:

```
HeapSort(A[1...n])
1  BuildHeap
2  for i ← n downto 2
3      swap(A[1], A[i])
4      size ← size − 1
5      Heapify(A, 1)
```

Коректността следва от верността на следната инвариантa: При всяко изпълнение на ред 2 масивът $A[1 \ldots size]$ е пирамида, а подмасивът $A[size+1 \ldots n]$ е сортиран и всичките му елементи са по-големи или равни на елементите на подмасива $A[1 \ldots size]$.

Сложността на $HeapSort$ е $\Theta(n \lg n)$. Той не използва допълнителна памет (освен малък брой локални променливи). Алгоритъмът не е стабилна сортировка, при всяка конкретна реализация на $BuildHeap$ и $Heapify$ лесно се строи контрапример.

1.4 QuickSort

Алгоритъмът QuickSort е предложен от C. A. R. Hoare през 1962г. Той, както и MergeSort е реализация на схемата 'Разделяй и владей', но при фазата на разделяне се прилага по-фина идея:

Разделяне: избира се специален елемент *splitter* (разделител) и масивът, който сортираме се пренарежда така, че отляво да са елементите по-малки от разделителя, после следва разделителят, а надясно от него се разполагат останалите по-големи или равни на *splitter* елементи. Това разместване за подинтервал на масива се извършва от процедурата *Partition*, описана по-долу.

Решаване на подзадачите: При тази фаза малките елементи наляво от разделителя и големите, които са надясно от него се сортират поотделно с рекурсивно извикване на процедурата *QuickSort*, описна по-долу, която също обработва подинтервал на масива, който сортираме. Сортировката на целия масив се извършва с извикване $QuickSort(A, 1, n)$.

Обединяване: Тази фаза не е необходима, тъй като редицата от малки сортирани елементи, следвани от разделителя, следван от големите сортирани елементи е окончателната, сортирана подредба на елементите на входния масив.

```
PARTITION(A[1,2,...,n]: array; l,h: indices in A)
 1  splitter ← A[h]
 2  p ← l
 3  for i ← l to h − 1
 4      if A[i] < splitter
 5          swap(A[i], A[p])
 6          p ← p + 1
 7  swap(A[p], A[h])
 8  return p
```

Коректност: Инвариантa на цикъла: При всяко достигане на ред 3, подмасивът $A[l, p-1]$ съдържа елементи, по-малки от разделителя, а подмасивът $A[p, i-1]$ - по-големи или равни.

Доказването на верността на инвариантната формула ще оставим на читателя, остава само да отбележим, че след приключване на цикъла, *swap*-ът на ред 7 ще разположи разделителя точно между малките и големите елементи на обработвания подмасив. □

Очевидно сложността на *Partition* е линейна спрямо размера на подмасива, който обработва, поради наличието на единствен цикъл в който се вършат краен брой сметки.

Тази версия на *Partition* е предложена от Nico Lomuto, през 1984г. Тя е по-кратка и удобна за доказване на коректност, но по-бавна (изпълнява повече команди *swap*) от оригиналната версия на Hoare.

```
QUICKSORT(A[1,2,...,n]: array; l,h: indices in A)
 1  if l < h
 2      m ← Partition(A, l, h)
 3      QuickSort(A, l, m − 1)
 4      QuickSort(A, m + 1, h)
```

Коректност: Трябва да докажем, че алгоритъмът изложен по-горе сортира подмасива зададен от границите l и h.

Доказателството може да се извърши по индукция спрямо размера на разглеждания подмасив $k = h-l+1$ и разсъждения за броя и разположението на инверсиите след всяка стъпка, а именно:

1. След изпълнението на *Partition* няма да има инверсии между малък елемент наляво от разделителя и голям, който е надясно от него.

2. Рекурсивните извиквания на *QuickSort* пък ще унищожат инверсиите между двойките малки елементи отляво и между двойките големи отдясно, заради индукционното предположение че програмата ще работи коректно за по-къси подмасиви.

3. Като резултат в края на алгоритъма всички инверсии ще изчезнат, т.е. подмасивът ще е сортиран. □

Сложност: QuickSort в лошия случай има сложност $O(n^2)$ - може да се случи *splitter* винаги да има екстремална стойност за интервала, който се разделя на две от процедурата *Partition*.

Например когато входният масив е сортиран, *Partition* винаги ще избира за разделител най-големият елемент в подмасива, който обработва. Тогава големите елементи ще са 0, а малките с 1 по-малко от размера на подмасива, и рекурентната формула за сложността ще бъде $T(n) = T(n-1)+cn$, с решение $O(n^2)$. Същото ще се случи и когато входният масив е сортиран в намалящ ред и изобщо в случаите когато броят на инверсиите в него е много малък или пък много голям.

В този смисъл интересен е анализът на средната сложност (average case complexity).

Нека означим с $T(n)$ средната сложност. Ако *Partition* разделя масива A в позиция k, ще имаме равенството $T(n) = T(k-1) + T(n-k) + O(n)$. Да представим $O(n)$ във вида $c(n+1)$.

Понеже k ще обхожда с равна вероятност $1/n$ стойностите от 1 до n, математическото очакване за $T(n)$ ще бъде:

$$T(n) = \frac{2}{n}\sum_{i=1}^{n-1} T(i) + c(n+1) \tag{1.1}$$

Умножаваме 1.1 по n:

$$nT(n) = 2\sum_{i=1}^{n-1} T(i) + cn(n+1) \tag{1.2}$$

Записваме 1.2 за $n-1$:

$$(n-1)T(n-1) = 2\sum_{i=1}^{n-2} T(i) + cn(n-1) \tag{1.3}$$

Сега изваждаме 1.3 от 1.2:

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn \tag{1.4}$$

След прехвърляне на $T(n-1)$ отдясно и делене на n получаваме:

$$T(n) = \frac{n+1}{n}T(n-1) + 2c \tag{1.5}$$

Решаваме 1.5 със заместване:

$$
\begin{aligned}
T(n) &= \frac{n+1}{n}(\frac{n}{n-1}T(n-2)+2c)+2c \\
&= \frac{n+1}{n-1}T(n-2)+2c(1+\frac{n+1}{n}) \\
&= \frac{n+1}{n-1}T(n-2)+2c(n+1)(\frac{1}{n+1}+\frac{1}{n}) \\
&= \frac{n+1}{n-2}T(n-3)+2c(n+1)(\frac{1}{n+1}+\frac{1}{n}+\frac{1}{n-1}) \\
&\dots \\
T(n) &= (n+1)T(0)+2c(n+1)\sum_{i=1}^{n+1}\frac{1}{i} \\
T(n) &= (n+1)T(0)+2c(n+1)H_{n+1} \\
T(n) &= (n+1)T(0)+2c(n+1)\Theta(\lg(n)) \\
T(n) &= \Theta(n\lg(n))
\end{aligned}
$$

По-горе с H_n е означена сумата на хармоничния ред.

□

1.4.1 Предимства и недостатъци на QuickSort

На пръв поглед изглежда, че QuickSort е по-слаб алгоритъм от MergeSort и HeapSort. Основанията за това са:

1. MergeSort и HeapSort имат гарантирана сложност $O(n\lg n)$, а QuickSort достига тази сложност в средния случай, но в най-лошия случай сложността му е $O(n^2)$.

2. MergeSort е стабилна сортировка, а HeapSort не използва допълнителна памет. QuickSort не е стабилна сортировка и използва памет $O(n)$ в лошия случай (тогава ще има n рекурсивни извиквания, които заемат памет от стека).

Някои автори (най-вече Robert Sedgewick [1]) твърдят, че QuickSort има по-малка константа при асимптотичната оценка, от MergeSort и HeapSort.

Причините за по-бавната работа на MergeSort и HeapSort, както и възможни техники за подобряването на тези два алгоритъма ще обсъдим на друго място (*вероятно в предните две секции, които още не са написани !*).

HeapSort при работа с няколко нива на паметта (бърза и бавна) е по-слаб от другите два алгоритъма поради факта, че непрекъснато обработва разни елементи по цялата дължина на масива, а MergeSort и QuickSort обработват къси подмасиви накуп, което ги съгласува добре с алгоритмите за работа с cache/swap памети и буфери.

Освен това недостатъците на QuickSort могат да бъдат коригирани (някои напълно, други частично), както ще покажем в следващата секция.

1.4.2 Подобрения на QuickSort

1. Ускоряване на разделянето:

Първото практическо подобрение не засяга недостатък на алгоритъма, то е всъщност оригиналната процедура по разделяне на Hoare:

HOARE_PARTITION($A[1,2,\dots,n]$: array; l,h: indices in A)

```
1  splitter ← A[h]
2  i ← l − 1
3  j ← h
4  while TRUE do
5      repeat
6          i ← i + 1
7      until A[i] ≥ splitter
8      repeat
9          j ← j − 1
10     until (A[j] < splitter) ∨ (j ≤ i)
11     if i < j
12         swap(A[i], A[j])
13     else swap(A[i], A[h])
14          return i
```

Анализът за коректност на горната процедура е по-сложен от версията на Lomuto и ще я оставим на читателя.

Броят сравнения на елементи да двата алгоритъма за разделяне е приблизително равен. Алгоритъмът на Hoare прави средно около 1.5 пъти по-малко сравнения и увеличавания/намаляния на индекси. Броят *swap*-ове при Lomuto съвпада с броя на малките елементи, при Hoare това е броят на големите елементи, които заемат първоначално местата на малки елементи, т.е. средно е 2-пъти по-бърз по този показател.

2. Намаляне на използваната памет:

Както беше отбелязано по-горе, в лошия случай броят на рекурсивните извиквания може да достигне $O(n)$, съответно допълнително използваната памет става $O(n)$, като става дума за стекова памет.

В някои операционни системи (или софтуерни среди в по-общия случай) може да има ограничения за размера на стека. В такава среда попадането в лошия случай не само ще увеличи използваната памет, а може да предизвика препълване на стека и прекъсване на изпълнението на алгоритъма.

Решението на този проблем е използването на опашкова рекурсия (tail-рекурсия). Това е техника за замяна на рекурсивно извикване с цикъл, когато рекурсивното извикване е в края (опашката) на дефиницията на процедура или функция. Някои компилатори и интерпретатори автоматизират такава замяна, но тази възможност е свързана с изисквания към стека и командите за управление на конкретната архитектура на изчислителната система.

Няма да обсъждаме в детайли техниката на опашковата рекурсия, само ще покажем как тя се прилага към алгоритъма QuickSort. Алгоритъмът завършва с рекурсивно обръщение към себе си, тоест, налице са условия за прилагане на техниката.

Ето как изглежда модифицирана версия на алгоритъма:

```
TailRec0_QuickSort(A[1, 2, ..., n]: array; l, h: indices in A)
1  while l < h do
2      m ← Partition(A, l, h)
3      TailRec0_QuickSort(A, l, m − 1)
4      l ← m + 1
```

Новата версия TailRec0_QuickSort извършва същите изчисления като оригиналната (докажете това!), но прави рекурсивно извикване само при сортирането на левия подинтервал (елементите

по-малки от разделителя *splitter*).

Все още съществува риск да попаднем в лош случай, при който десния интервал (който обработваме в цикъл) да се случва да е кратък, а левият (който обработваме рекурсивно) да е дълъг и да получим много нива на рекурсивни извиквания.

Решението е рекурсивното обръщение да се прави към по-късия подинтервал, а по-дългият да се обработва в цикъл чрез механизма на опашковата рекурсия. Това е възможно, понеже няма значение в какъв ред сортираме малките и големите елементи след разделянето спрямо *splitter* и можем да разменяме реда на рекурсивните извиквания на QuickSort в оригиналния алгоритъм. Така при всяко рекурсивно извикване размерът на обработвания масив ще намалява поне наполовина и дълбочината на използвания стек ще бъде $O(\lg(n))$.

Ето модификацията, която гарантирано използва $O(\lg(n))$ допълнителна памет:

```
TailRec_QuickSort(A[1,2,...,n]: array; l,h: indices in A)
 1  while l < h do
 2      m ← Partition(A,l,h)
 3      if m − l < h − m
 4          TailRec_QuickSort(A,l,m − 1)
 5          l ← m + 1
 6      else TailRec_QuickSort(A,m + 1,h)
 7           h ← m − 1
```

3. Случаен избор на разделител:

Лошият случай при работата на алгоритъма QuickSort възниква когато избраният разделител *splitter* има екстремална или близка до екстремална стойност (т.е. той е най-малкият, най-големият или близък до тях елемент в изследвания интервал). Тогава интервалът се разделя на една много малка и друга много голяма част, и при серия от такива *лоши* разделяния сложността нараства до $\Theta(n^2)$.

На практика често се случва потребителят да подаде към сортиращият алгоритъм вече сортиран масив. Но това е точно най-лошият случай за алгоритъма QuickSort, в неговата основна реализация, изложена в началото на тази глава.

Може да приложим някаква стратегия за избягване на подобна неприятна ситуация, примерно функцията Partition да избира за *splitter* не краен, а среден елемент. Тогава възниква друга опасност: хакер, запознат с детайли на нашата реализация да подаде като входни данни така подреден масив, че новата ни функция Partition винаги да избира за *splitter* максиманият елемент на обработвания интервал и отново да ни вкара в лошия случай.

Едно от възможните средства за предотвратяване на неприятните ситуации, описани по-горе, е да избираме *случаен* елемент за разделител. Това може да стане във функцията Partition – избираме случаен индекс i ($l \leq i \leq h$), и работим с разделител $splitter = A[i]$. Друг удобен метод е преди началото на сортировката да разбъркаме случайно елементите на входния масив A.

Техниката на избор на случен разделител не намалява вероятността за влизане в лош случай, тя променя лошия случай от предварително сортиран масив в друг (случайно нареден). Тя предотвратява възможността за хакерска атака, стига хакерът да не знае каква случайна поредица използваме за избора на разделител (или за разбъркването на масива в началото).

4. По-добър разделител.

Медиана на масив от числа ще наричаме този елемент, който заема средната позиция в сортирания масив, т.е. елемент, който е по-голям или равен на половината елементи на масива и

по-малък или равен на другата половина.

Алгоритъмът QuickSort достига най-голяма ефективност когато разделителят е близък до медианата. Струва си, когато входният масив е сравнително голям, да вземем някакво количество негови елементи, да намерим тяхната медиана и да я изберем за разделител. Има шанс така изчисления разделител да е по-близо до същинската медиана и по-далеч от краищата на масива.

Най-простият случай на такава модификация (нарича се *среден от 3*) е да се вземат 3 елемента (примерно $A[l], A[\lfloor (l+h)/2 \rfloor], A[h]$, или пък 3 случайно избрани) и средният по големина да се ползва за разделител.

При обикновен избор на разделител той е с равна вероятност кой да е от елементите на входния масив от n елемента, а математическото очакване за размерите на подинтервалите (малък и голям) след разделянето е съответно $\frac{1}{4}n$ и $\frac{3}{4}n$.

При избор на разделител по метода *среден от 3* съответните очаквания са $\frac{5}{16}n$ и $\frac{11}{16}n$, което приближава малко (на $\frac{1}{16}n$) разделителя до медианата.

Вероятността обикновения разделител да е попадне след разделянето в позиция $1 \dots i$ (да е в левия край на масива, близо до минималната стойност) е $Pr[Partition(A, 1, n) \leq i] = \frac{i}{n}$. При метода *среден от 3* съответната вероятност е ограничена от $c(\frac{i}{n})^2$ за някаква константа c, което значително намалява шанса да се получи много лошо разделяне. Същите разсъждения важат и за десния край на масива.

Още по-добри резултати ще се получат, ако се избира раделител от по-голяма извадка (примерно среден от 5 или 7), но изчисляването на такъв разделител изисква допълнително време и може да се ползва само когато входния интервал е голям.

5. Малките подмасиви се сортират с *InsertionSort*.

Когато в някой подмасив останат около 10 елемента, бързите алгоритми (QuickSort, MergeSort, HeapSort) са по-неефективни от *InsertionSort*, тъй като правят много рекурсивни извиквания към себе си (QuickSort, MergeSort), или разместват неефективно елементите си (HeapSort).

Алгоритъмът *InsertionSort* е малко по-бърз за малък брой елементи и QuickSort може да се модифицира така, че когато обработваният интервал стане по-къс от някаква константа (Седжуик препоръчва 10-16), той да бъде сортиран с *InsertionSort*.

6. Комбиниране с друг алгоритъм:

Използването на случаен разделител помага на алгоритъма QuickSort да избегне влизането в лошия случай при някои обичайни случаи на входни данни или при хакерска злоупотреба, но нарушава свойството определеност[1] (детерминираност) на алгоритъма.

Пълното избягване на лошия случай и запазването на опредлеността се постига чрез комбинирането на QuickSort с друг алгоритъм (на практика се ползва HeapSort). В началото сортировката започва с алгоритъма QuickSort, като се следи размера на изследвания интервал и дълбочината на рекурсията (или броя на итерациите при прилагане на опашкова рекурсия). Ако размерът на интервала не намалява значително при нарастване на дълбочината на рекурсия, следва, че сме влезли в лошия случай – тогава оставащият интервал се сортира с HeapSort. Подобна схема се прилага в съвременната алгоритмична библиотека STL за езика C++.

[1] Определеността изисква при всяко стартиране с едни и същи данни алгоритъмът да дава един и същ резултат и да провежда изчисленията по еднакъв начин. В нашият случай QuickSort винаги ще подрежда коректно входния масив, но изборът на случаен разделител ще доведе до различни времена за изпълнение и до евентуално разместване в изходния файл на елементи с еднакъв ключ.

Библиография

[1] Robert Sedgewick: Implementing Quicksort Programs, Communications of the ACM, October 1978 Volume 21 Number 10

[2] Herbert S. Wilf: Algorithms and Complexity, Internet Edition, 1994