Европейски съюз

ОП "Развитие на човешките ресурси"
Договор BG051PO001-4.3.04-0018
Разработване на програми за електронни форми на дистанционно обучение във Факултета по математика и информатика

Европейски социален фонд

Трета лекция по ДАА на втори поток КН
Двоична Пирамида. HeapSort. Приоритетна Опашка.

17 март 2014

Абстракт

Въвеждаме понятието попълнено двоично дърво и двоична пирамида. Показваме как се строи двоична пирамида, по наивен начин във време $\Theta(n \lg n)$ и бързото построяване във време $\Theta(n)$ на Floyd. Демонстрираме сортиращия алгоритъм HeapSort и го анализираме. Въвеждаме понятието приоритетна опашка като абстрактен тип данни и разискваме имплементация на приоритетна опашка чрез двоична пирамида.

Съдържание

1 Двоични дървета и пирамиди

1.1 Попълнено двоично дърво

Тъй като теорията на графите е сравнително нова област, няма универсално приета пълна единна терминология и всяко сериозно изложение, използващо графови понятия, трябва да започва с изчерпателни определения, за да е ясно точно какво се има предвид. Терминологията на английски ще изложим според американския Национален Институт за Стандарти и Технологии NIST. Техните определения за дървета са изложени на тази страница. Не допускаме празни дървета (без върхове). Разглеждаме само коренови дървета, така че когато кажем "дърво" (tree), имаме предвид кореново дърво (rooted tree). В контекста на структури от данни по правило дърветата

са *наредени*[†] (ordered trees), което означава, че децата на вътрешните върхове са наредени с тотална наредба. Следните две дървета T_1 и T_2 са различни като наредени дървета (наредбата е отляво надясно), но са едно и също дърво, ако ги разглеждаме като коренови дървета:

Ако T е кореново дърво с корен r, за всеки връх u в него с $T[\![u]\!]$ бележим поддървото, вкоренено в u. Формално,

- ако $u = r$, то $T[\![u]\!]$ е самото T,
- в противен случай, $T[\![u]\!]$ е поддървото, което се получава при премахване на реброто (u, v) от T и което съдържа u, където v е върхът-родител на на u в T.

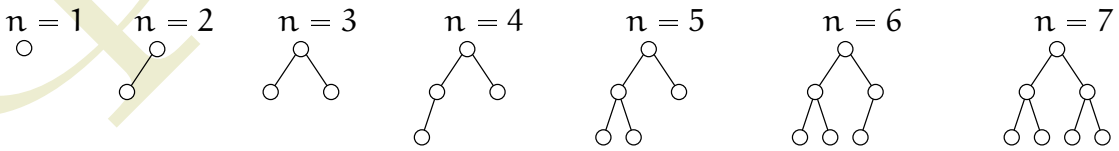
Височина на връх u в кореново дърво е максималното разстояние между u и кое да е листо в $T[\![u]\!]$. *Височина на дървото* е височината на корена. *Дълбочина на връх* u в кореново дърво е разстоянието между него и корена. *Ниво* в дърво е множеството от всички върхове, които са на една и съща дълбочина.

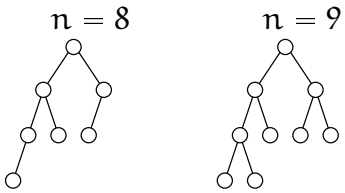
Двоично дърво е кореново дърво (наредено или не), в което всеки връх има не повече от две деца.

Определение 1 (пълно двоично дърво, не-индуктивно определение)**.** Пълно двоично дърво *(complete binary tree) е наредено двоично дърво, в което:*

- *всички листа са или на едно ниво, или на две нива, чиито височини имат разлика единица*
- *и освен това, ако листата са на две нива, листата на последното ниво са максимално вляво.* □

За всяко $n \in \mathbb{N}^+$ има едно единствено пълно дърво с n върха:





Съвършено двоично дърво (perfect binary tree) е пълно двоично дърво, в което всички листа са на едно и също ниво [NIS].

Съществува еквивалентно индуктивно определение на пълно двоично дърво

Определение 2 (пълно двоично дърво, индуктивно определение)**.** *Пълно двоично дърво е всяко дърво, което може да бъде получено от краен брой прилагания на следните конструкции:*

1. $(\{u\}, \varnothing)$ *е пълно двоично дърво с корен* u*, множество от листа* $\{u\}$*, множество от вътрешни върхове* $\varnothing$ *и височина* 0*.*

[†]За разлика от "чистата" теория на графите, в която по правило кореновите дървета не са наредени.

2. $(\{u, v\}, \{(u, v)\})$ *е попълнено двоично дърво с корен* u*, множество от листа* $\{v\}$*, множество от вътрешни върхове* $\{u\}$ *и височина* 1*.* v *е ляво дете на* u*.*

3. *Нека* T' *е съвършено кореново дърво с корен* u'*, множество от листа* W'*, и височина* $h - 1 \geqslant 0$*. Нека* T'' *е попълнено кореново дърво с корен* u''*, множество от листа* W''*, и същата височина височина* $h - 1$*. Нека* r *е връх извън* T' *и* T''*. Тогава*

$$\big(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\}\big),$$

където u' *е ляво дете на* r*, а* u'' *е дясно дете на* r*, е попълнено кореново дърво с корен* r*, множество от листа* $W' \cup W''$ *и височина* h*.*

4. *Нека* T' *е попълнено кореново дърво с корен* u'*, множество от листа* W'*, и височина* $h - 1 \geqslant 1$*. Нека* T'' *е съвършено кореново дърво с корен* u''*, множество от листа* W''*, и височина* $h - 2$*. Нека* r *е връх извън* T' *и* T''*. Тогава*

$$\big(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\}\big),$$

където u' *е ляво дете на* r*, а* u'' *е дясно дете на* r*, е попълнено кореново дърво с корен* r*, множество от листа* $W' \cup W''$ *и височина* h*.*

Доказателството за еквивалентност на Определение 1 и Определение 2 остават за читателя.

Както обикновено, пишейки $\lg n$, имаме предвид $\log_2 n$.

Лема 1. *За всяко попълнено двоично дърво с* n *върха, за височината* h *е изпълнено* $h = \lfloor \lg n \rfloor$*.*

Доказателство: Чрез структурна индукция съгласно Определение 2. В базовите случаи 1 и 2 твърдението е вярно: имаме $0 = \lfloor \lg 1 \rfloor$ и $1 = \lfloor \lg 2 \rfloor$. ✓

В случай 3, нека T' има p върха и T'' има q върха. Съгласно индуктивното предположение, $h - 1 = \lfloor \lg p \rfloor$ и $h - 1 = \lfloor \lg q \rfloor$. Но $\lfloor x \rfloor$ е най-голямото **цяло число, ненадхвърлящо** x. Можем да твърдим, че:

$$h - 1 \leqslant \lg p < h$$
$$h - 1 \leqslant \lg q < h$$

Тогава,

$$h \leqslant (\lg p) + 1 < h + 1$$
$$h \leqslant (\lg q) + 1 < h + 1$$

Другояче казано,

$$h \leqslant \lg 2p < h + 1$$
$$h \leqslant \lg 2q < h + 1$$

Тъй като $\min\{2p, 2q\} \leqslant p + q \leqslant \max\{2p, 2q\}$, можем да твърдим, че

$$h \leqslant \lg(p + q) < h + 1$$

Но щом $h - 1 \leqslant \lg p < h$, то $p < 2^h$. Аналогично, $q < 2^h$. Тогава $p + q < 2^{h+1}$. Нещо повече, $p + q + 1 < 2^{h+1}$. Тогава $\lg(p + q + 1) < h + 1$. Следователно,

$$h \leqslant \lg(p + q + 1) < h + 1$$

Но $n = p + q + 1$. Тогава,

$$h \leqslant \lg n < h + 1$$

Следователно, $h = \lfloor \lg n \rfloor$. ✓

В случай 4, нека T' има p върха и T'' има q върха. Съгласно индуктивното предположение, $h - 1 = \lfloor \lg p \rfloor$ и $h - 2 = \lfloor \lg q \rfloor$. Но q е число от вида $2^k - 1$ за някое $k \geqslant 0$, защото T'' е съвършено дърво. Тогава $q + 1 = 2^k$.

Следователно, $\lfloor \lg (q+1) \rfloor = \lfloor \lg q \rfloor + 1$, така че $h - 1 = \lfloor \lg (q+1) \rfloor$. С разсъждения, аналогични на предния случай, получаваме

$$h - 1 \leqslant \lg p < h$$
$$h - 1 \leqslant \lg (q+1) < h$$

Тогава,

$$h \leqslant \lg p + 1 < h + 1$$
$$h \leqslant \lg (q+1) + 1 < h + 1$$

Другояче казано,

$$h \leqslant \lg 2p < h + 1$$
$$h < \lg (2q+2) < h + 1$$

Тъй като $\min\{2p, 2q+2\} \leqslant p + q + 1 \leqslant \max\{2p, 2q+2\}$, можем да твърдим, че

$$h \leqslant \lg (p+q+1) < h + 1$$

Но $n = p + q + 1$. Тогава,

$$h \leqslant \lg n < h + 1$$

Следователно, $h = \lfloor \lg n \rfloor$. ✓ □

Лема 2. *Всяко попълнено двоично дърво* T *има точно* $\lceil \frac{n}{2} \rceil$ *листа.*

Доказателство: Чрез структурна индукция съгласно Определение 2. За двата базови случаи твърдението е вярно: в случай 1, $n = 1$ и наистина има $\lceil \frac{1}{2} \rceil = 1$ листа, а в случай 2, $n = 2$ и наистина има $\lceil \frac{2}{2} \rceil = 1$ листа. ✓

Нека T е конструирано чрез случай 3. Нека T′ има p върха и T″ има q върха. Очевидно $n = p + q + 1$. И T′, и T″ са попълнени дървета и индуктивното предположение тях него казват, че те имат съответно $\lceil \frac{p}{2} \rceil$ и $\lceil \frac{q}{2} \rceil$ листа. Наготово използваме факта, че T′ има нечетен брой върхове, понеже е съвършено двоично дърво. Нека $p = 2k + 1$. Съгласно Определение 2, множеството от листата на T се разбива на множествата от листата на T′ и T″, така че броят на листата на T е:

$$\left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil = \left\lceil \frac{2k+1}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil = k + 1 + \left\lceil \frac{q}{2} \right\rceil = \left\lceil k + 1 + \frac{q}{2} \right\rceil = \left\lceil \frac{2k+2+q}{2} \right\rceil = \left\lceil \frac{2k+1+q+1}{2} \right\rceil$$
$$= \left\lceil \frac{p+q+1}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil \quad ✓$$

Нека T е конструирано чрез случай 4. Нека T′ има p върха и T″ има q върха. Очевидно $n = p + q + 1$. И T′, и T″ са попълнени дървета и индуктивното предположение тях него казват, че те имат съответно $\lceil \frac{p}{2} \rceil$ и $\lceil \frac{q}{2} \rceil$ листа. Наготово използваме факта, че T″ има нечетен брой върхове, понеже е съвършено двоично дърво. Нека $q = 2k + 1$. Съгласно Определение 2, множеството от листата на T се разбива на множествата от листата на T′ и T″, така че броят на листата на T е:

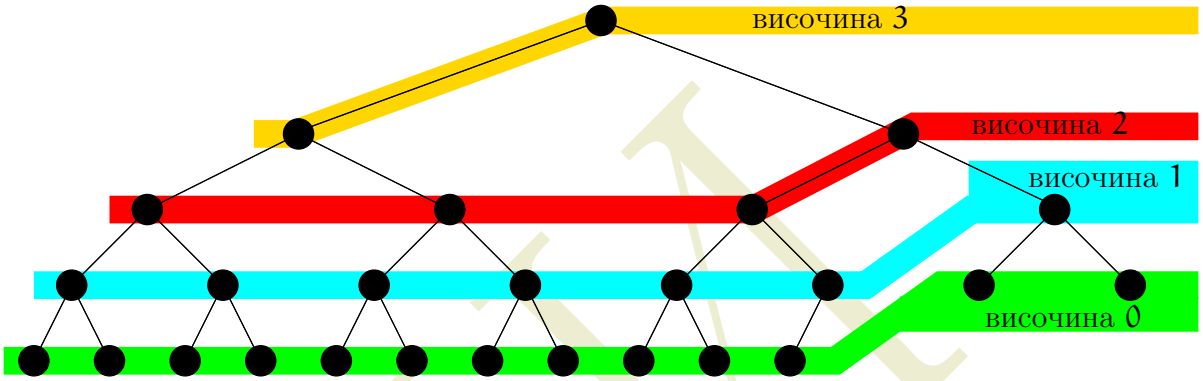
$$\left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil = \left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{2k+1}{2} \right\rceil = \left\lceil \frac{p}{2} \right\rceil + k + 1 = \left\lceil \frac{p}{2} + k + 1 \right\rceil = \left\lceil \frac{p+2k+2}{2} \right\rceil = \left\lceil \frac{p+2k+1+1}{2} \right\rceil$$
$$= \left\lceil \frac{p+q+1}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil \quad ✓$$

□

Следствие 1. *Вътрешните върхове на произволно попълнено двоично дърво с* n *върха са* $\lfloor \frac{n}{2} \rfloor$.

Доказателство: Следва веднага от Лема 2 и факта, че $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = n$ за всяко цяло n. □

Следните равенства са от книгата "Конкретна Математика" (Concrete Mathematics) [GKP94, стр. 71].



Фигура 1: Височините на върховете в попълнено двоично дърво T.

Лема 3. *Нека $f(x)$ е произволна реална монотонно растяща функция със свойството:*

$$f(x) \text{ е цяло} \rightarrow x \text{ е цяло}$$

Тогава,

$$\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor \quad \text{и} \quad \lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil$$

□

От Лема 3 веднага имаме това следствие.

Следствие 2.

$$\forall x \in \mathbb{R}^+ \; \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor x \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{b} \right\rfloor \quad \text{и} \quad \left\lceil \frac{\lceil x \rceil}{b} \right\rceil = \left\lceil \frac{x}{b} \right\rceil \right)$$

Доказателство:
Прилагаме Лема 3 с $f(x) = \frac{x}{b}$. □

Равенствата от Лема 3 са дадени в [CLRS09] без никакво доказателство. Тук ги доказваме, ползвайки Следствие 2.

Следствие 3.

$$\forall x \in \mathbb{R}^+ \; \forall a \in \mathbb{N}^+ \; \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor \frac{x}{a} \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor \quad \text{и} \quad \left\lceil \frac{\lceil \frac{x}{a} \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil \right)$$

Доказателство:
Прилагаме Следствие 2 с $\frac{x}{a}$ наместо x. □

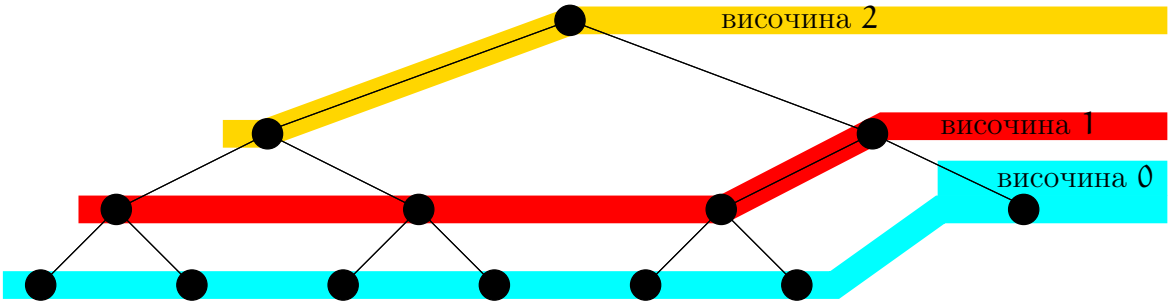
Лема 4. *Във всяко попълнено двоично дърво T с n върха има точно $\left\lfloor \frac{n}{2^k} \right\rfloor$ върха с височина $\geqslant k$.*

Доказателство:
Да си припомним, че *височина на връх* u е максималното разтояние от него до кое да е листо в поддървото $T[\![u]\!]$. Пример за височините на върхове има на Фигура 1. Доказателството е по индукция по k.

База. $k = 0$. Върховете с височина $\geqslant 0$ са точно върховете на T, които на n на брой. Забележете, че $n = \left\lfloor \frac{n}{2^0} \right\rfloor$. ✓

Индуктивно предположение. Нека твърдението е в сила за някаква височина k, която не е максималната.

Индуктивна стъпка. Да изтрием всички върхове с височина $< k$ от T. Нека полученото дърво е T' и нека n' е броят на неговите върхове. Листата на T', тоест върховете с височина 0 в T', очевидно са точно върховете с височина k в T. Всички върхове в T' са точно върховете с височина $\geqslant k$ в T. Съгласно индуктивното предположение, $n' = \left\lfloor \frac{n}{2^k} \right\rfloor$. Примерно, да разгледаме дървото T от Фигура 1 с $k = 1$: след изтриването на всички върхове с височина < 1 получаваме дървотоT' от Фигура 2.



Фигура 2: Попълненото дърво T', което се получава от дървото T от Фигура 1 след изтриване на всички върхове с височина < 1, тоест с височина 0, тоест листата. Това изтриване намалява с точно единица височините на останалите върхове.

Съгласно Следствие 1, има точно $\left\lfloor \frac{n'}{2} \right\rfloor$ вътрешни върхове в T'. Но вътрешните върхове на T' са точно върховете с височина $\geqslant k+1$ в T. Следователно, има $\left\lfloor \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rfloor$ върха с височина $\geqslant k+1$ в T. Съгласно Лема 3, $\left\lfloor \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rfloor = \left\lfloor \frac{n}{2 \times 2^k} \right\rfloor$, и очевидно $\left\lfloor \frac{n}{2 \times 2^k} \right\rfloor = \left\lfloor \frac{n}{2^{k+1}} \right\rfloor$. □

Следствие 4. *Във всяко попълнено двоично дърво T с n върха има точно $\left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil$ върха с височина k.*

Доказателство:
Съгласно Лема 4, има точно $\left\lfloor \frac{n}{2^k} \right\rfloor$ върха с височина $\geqslant k$. Върховете с височина k са точно листата в поддървото на T, индуцирано от върховете с височина $\geqslant k$. Съгласно Лема 2, тези листа са точно $\left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil$. □

1.2 Адреси в попълнено двоично дърво

Следното определение задава процедура, която дава адреси-стрингове на върховете на попълнено двоично дърво, използвайки Определение 2. С "ϵ" означаваме празния стринг.

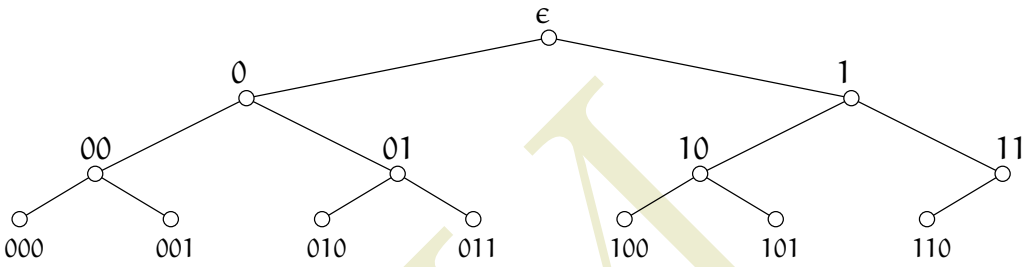
Определение 3 (адреси на върхове в попълнено двоично дърво)**.** *Всеки връх x в попълненото дърво има* адрес, *който бележим с $addr(x)$. Адресите се конструират от следната процедура от две фази.*

фаза I: *В базовия случай 1, $addr(u) = \epsilon$. В базовия случай 2, $addr(u) = \epsilon$ и $addr(v) = 0$. В случаи 3 и 4:*

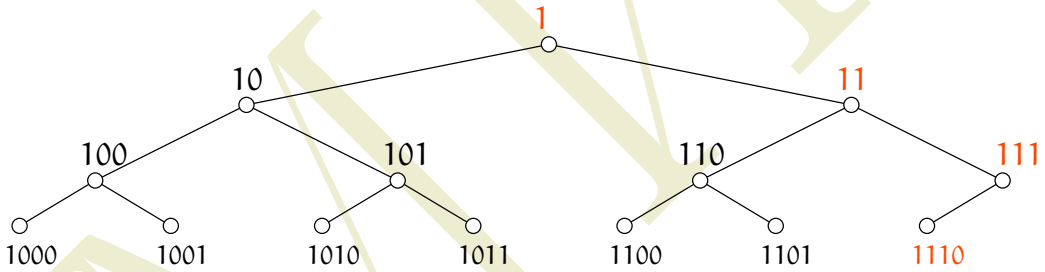
- *$addr(u) = \epsilon$,*
- *за всеки връх x от T', новият адрес на x се получава от стария с конкатенация на нула в края.*
- *за всеки връх x от T'', новият адрес на x се получава от стария с конкатенация на единица в края.*

фаза II: *За всеки връх x от T, новият адрес се получава от стария чрез конкатенация на единица в началото.*
□

Ето пример за конструиране на адресите от **фаза I** за попълненото дърво с 14 върха:



Продължавайки със същия пример, ето окончателните адреси след **фаза II**:



Според Определение 3 адресите на върховете в попълнените двоични дървета са стрингове. В изложението надолу понякога ще злоупотребяваме с това и ще третираме адресите като числа—а именно, числата, записани с тези стрингове.

Лема 5. *Нека* T *е произволно попълнено двоично дърво и* u *е произволен връх в него. Ако* u *не е листо:*

- *ако* u *има две деца* v *и* w*, където* v *е лявото дете, в сила е* $addr(v) = 2\times addr(u)$ *и* $addr(w) = 2\times addr(u)+1$.
- *ако* u *има точно едно дете* v*, в сила е* $addr(v) = 2 \times addr(u)$.

Ако u *не е корен и* v *е родителят на* u*, то* $addr(v) = \left\lfloor \frac{addr(u)}{2} \right\rfloor$. □

Доказателство, част I: Първо ще докажем твърдението за адресите на децата. Ще докажем твърдението по дълбочината на u.

Базовият случай е: u има дълбочина нула, тоест u е коренът. Ако u има две деца v и w, като v е лявото, $\text{addr}(u) = 1$, $\text{addr}(v) = 10$ и $\text{addr}(w) = 11$, така че твърдението е вярно. Ако u има точно едно дете, очевидно става дума за дърво, построено чрез базова конструкция 2 на Определение 2, и съгласно Определение 3, $\text{addr}(u) = 1$ и $\text{addr}(v) = 10$, следователно твърдението пак е вярно.

Нека u има дълбочина d и u има поне едно дете. Ако u има точно две деца v и w, като v е лявото, $\text{addr}(v) = \text{addr}(u)0$ и $\text{addr}(v) = \text{addr}(u)1$, следователно твърдението е вярно, когато разглеждаме адресите като числа, записани в двоична позиционна бройна система.

Доказателство, част II: Ще докажем твърдението за адреса на родителя на всеки връх, който не е коренът. Но това твърдение е очевидно, имайки предвид **част I**. Адресът на u, изразен чрез адресите на децата си, е

$$\text{addr}(u) = \frac{\text{addr}(v)}{2} \quad \text{и} \quad \text{addr}(u) = \frac{\text{addr}(w) - 1}{2}$$

като, ако u има само едно дете, очевидно само първият израз е в сила. Тъй като $\frac{\text{addr}(v)}{2}$ е задължително четно, а $\frac{\text{addr}(w)}{2}$, ако съществува, е задължително нечетно, исканото твърдение следва веднага. □

За всяко $n \in \mathbb{N}^+$, $\text{bin}(n)$ означава стринга-представяне на n в двоична позиционна бройна система (с водеща единица). Нека $n, k \in \mathbb{N}^+$ и $m = \lfloor \log_2 n \rfloor + 1$. Тогава $\text{bin}_k(n)$ се дефинира така:

$$\text{bin}_k(n) = \begin{cases} 0^{k-m}\,\text{bin}(n), & \text{ако } k > m \\ \text{bin}(n), & \text{ако } k = m \\ \text{суфиксът с дължина } k \text{ на } \text{bin}(n), & \text{ако } k < m. \end{cases}$$

Забележете, че в случая m е дължината на представянето на n в двоична позиционна бройна система. Също така забележете, че $\mathsf{bin}(n)$ и $\mathsf{bin}_k(n)$ са стрингове над азбуката $\{0,1\}$, а не числа.

Нека x и y са произволни два върха от едно и също ниво и връх c е общият предшественик с най-малка дълбочина на x и y. Казваме, че x е вляво от y, ако x е връх от $T[\![a]\!]$ и y е връх от $T[\![b]\!]$, където a е лявото дете, а b е дясното дете на c.

Лема 6. *Нека $T = (V, E)$ е произволно попълнено двоично дърво с n върха. Нека височината на T е h. Нека V_d е подмножеството от върховете на дълбочина d, за $0 \leqslant d \leqslant h$. Тогава*

(а) За $0 \leqslant d \leqslant h-1$, множеството от адресите на върховете от V_d е множеството от всички булеви стрингове с дължина $d+1$ и водеща единица.

(б) Множеството от адресите на върховете от V_h е множеството от булевите стрингове с дължина $h+1$ и водеща единица от 10^h до $\mathsf{bin}(n)$ в лексикографската наредба.

(в) Нещо повече, във всяко ниво върховете са лексикографски сортирани по адреси отляво надясно в T.

Доказателство: Нека $\mathrm{addr}'(x)$ е адресът на x след **фаза I**, за всеки връх $x \in V$. В термините на $\mathrm{addr}'(x)$, твърдения а и б са еквивалентни съответно на:

(а') За $0 \leqslant d \leqslant h-1$, множеството от адресите на върховете от V_d е множеството от всички булеви стрингове с дължина d.

(б') Множеството от адресите на върховете от V_h е множеството от булевите стрингове с дължина h от 0^h до $\mathsf{bin}_h(n)$ в лексикографската наредба.

Това ще докажем чрез структурна индукция съгласно Определение 2 на попълнено дърво, използвайки Определение 3 на адреси на върховете на попълнено дърво.

База. В случай 1 на Определение 2, единственият връх има адрес ϵ. В случай 2,

- в ниво 0 единственият връх наистина има адрес ϵ, който има дължина 0,
- в последното ниво 1 наистина адресите са от 0 до $\mathsf{bin}_1(2) = 0$. ✓

Индуктивна стъпка Да разгледаме случай 3.

- На ниво 0, коренът r наистина получава адрес ϵ.
- Да разгледаме произволно ниво d, където $1 \leqslant d \leqslant h-1$. Тривиално е да се докаже, че върховете от ниво d в T са 2^d на брой и че тяхната наредба отляво надясно се състои от върховете на ниво $d-1$ в T', последвани от върховете от ниво $d-1$ на T''. Тъй като $0 \leqslant d-1 \leqslant h-2$, ниво $d-1$ не е последно ниво нито в T', нито в T''. Тогава, съгласно част а на индуктивното допускане, адресите на върховете от ниво $d-1$ и в T', и в T'' са булевите стрингове с дължина $d-1$, които и в T', и в T'' са сортирани отляво надясно. Очевидно след слагане на 0 вляво на всеки адрес от ниво $d-1$ в T' и на 1 вляво на всеки адрес от ниво $d-1$ в T'' получаваме—по отношение на T—всички булеви стрингове с дължина d, сортирани лексикографски.
- Да разгледаме последното ниво h в T. В случай 3 то се състои от всички 2^{h-1} върхове от ниво $h-1$ на T', последвани вдясно от върховете от ниво $h-1$ на T''.

  Да разгледаме последното ниво $h-1$ в T'. Нека T' има p върха и T'' има q върха. Съгласно част б на индуктивното предположение, отляво надясно адресите в ниво $h-1$ на T' са

$$\underbrace{00\ldots0}_{\text{дължина } h-1}, \quad \underbrace{00\ldots01}_{\text{дължина } h-1}, \ldots, \underbrace{\mathsf{bin}_{h-1}(p)}_{\text{дължина } h-1}$$

  Но $p = 2^{h-1+1} - 1 = 2^h - 1$, защото T' е съвършено дърво, така че $\mathsf{bin}_{h-1}(p) = 1^{h-1}$ единици. Тогава адресите в ниво $h-1$ на T' са, отляво надясно,

$$\underbrace{00\ldots0}_{\text{дължина } h-1}, \quad \underbrace{00\ldots01}_{\text{дължина } h-1}, \ldots, \underbrace{11\ldots1}_{\text{дължина } h-1}$$

тоест всички булеви стрингове с дължина $h-1$.

Да разгледаме последното ниво $h-1$ в T''. Съгласно част а на индуктивното предположение, отляво надясно адресите в ниво $h-1$ на T'' са

$$\underbrace{00\ldots0}_{\text{дължина } h-1}\ ,\quad \underbrace{00\ldots01}_{\text{дължина } h-1}\ ,\ldots,\ \underbrace{\mathrm{bin}_{h-1}(q)}_{\text{дължина } h-1}$$

Следователно, в цялото дърво T, адресите на върховете от ниво h са, отляво надясно

$$\underbrace{00\ldots0}_{\text{дължина } h}\ ,\quad \underbrace{00\ldots01}_{\text{дължина } h}\ ,\ldots,\ \underbrace{011\ldots1}_{\text{дължина } h}\ ,\ \underbrace{10\ldots0}_{\text{дължина } h}\ ,\quad \underbrace{10\ldots01}_{\text{дължина } h}\ ,\ldots,\ \underbrace{1\,\mathrm{bin}_{h-1}(q)}_{\text{дължина } h}$$

Тъй като $n = p + q + 1$ и $p = 2^h - 1$, то $n = 2^h + q$. Но тъй като $q \leqslant p$, дължината на $\mathrm{bin}(q)$ също е по-малка от h и $\mathrm{bin}_{h-1}(q)$ се получава от $\mathrm{bin}(q)$ с добавяне на $\geqslant 0$ на брой нули вляво. Имайки предвид, че $\mathrm{bin}(2^h) = 10^h$, заключаваме, че стрингът $1\,\mathrm{bin}_{1-h}(q)$ е равен на $\mathrm{bin}(n)$. Заключаваме, че адресите от последното ниво на T' отляво надясно, следвани от адресите от последното ниво на T'' отляво надясно, представляват последователността

$$\underbrace{00\ldots0}_{\text{дължина } h}\ ,\quad \underbrace{00\ldots01}_{\text{дължина } h}\ ,\ldots,\ \underbrace{011\ldots1}_{\text{дължина } h}\ ,\ \underbrace{10\ldots0}_{\text{дължина } h}\ ,\quad \underbrace{10\ldots01}_{\text{дължина } h}\ ,\ldots,\ \underbrace{\mathrm{bin}(n)}_{\text{дължина } h}$$

Да разгледаме случай 4.

- На ниво 0, коренът r наистина получава адрес ϵ.
- Да разгледаме произволно ниво d в T, където $1 \leqslant d \leqslant h-1$. Аналогично на предния случай, върховете от ниво d в T са 2^d на брой и тяхната наредба отляво надясно се състои от върховете на ниво $d-1$ в T', последвани от върховете от ниво $d-1$ на T''. Тъй като $0 \leqslant d-1 \leqslant h-2$, ниво $d-1$ не е последното ниво в T', но е последното ниво в съвършеното дърво T'' при $d-1 = h-2$.

  Адресите на върховете от ниво $d-1$ в T' са булевите стрингове с дължина $d-1$ и са сортирани отляво надясно съгласно част а на индуктивното предложение. Ако $d-1 < h-2$, адресите на на върховете от ниво $d-1$ в T'' са булевите стрингове с дължина $d-1$ и са сортирани отляво надясно съгласно част а на индуктивното предложение.

  Да разгледаме случая $d-1 = h-2$ по отношение на T''. Сега част б на индуктивното предположение е приложима. Съгласно нея, адресите на на върховете от ниво $d-1$ в T'' са булевите стрингове от 0^{h-2} до $\mathrm{bin}_{h-2}(q)$ в лексикографската наредба, където q е броят на върховете в T'', и те са сортирани отляво надясно лексикографски. Но тъй като T'' е съвършено дърво с височина $h-2$, вярно е, че $q = 2^{h-2+1} - 1 = 2^{h-1} - 1$, следователно $\mathrm{bin}_{h-2}(q) = 1^{d-1}$.

  Заключаваме, че за $1 \leqslant d \leqslant h-1$, адресите на върховете от ниво $d-1$ в T' са всички булеви стрингове с дължина $d-1$, сортирани отляво надясно, а също така адресите на върховете от ниво $d-1$ в T'' са всички булеви стрингове с дължина $d-1$ сортирани отляво надясно. Очевидно след слагане на 0 вляво на всеки адрес от ниво $d-1$ в T' и на 1 вляво на всеки адрес от ниво $d-1$ в T'' получаваме—по отношение на дървото T—всички булеви стрингове с дължина d, сортирани лексикографски отляво надясно.
- Да разгледаме последното ниво h в T. То е последното ниво $h-1$ на T', така че част б на индуктивното предположение е приложима. Съгласно нея, адресите на на върховете от ниво $h-1$ в T' са булевите стрингове от 0^{h-1} до $\mathrm{bin}_{h-1}(p)$, където p е броят на върховете в T', и те са сортирани отляво надясно лексикографски. Тогава във **фаза I** тези върхове получават адреси от 0^h до $0\mathrm{bin}_{h-1}(p)$, сортирани отляво надясно лексикографски.

  Да си припомним, че $n = p + q + 1$, където q е броят на върховете в съвършеното дърво T''. Тъй като T'' е съвършено и с височина $h-2$, вярно е, че $q = 2^{h-1} - 1$, следователно $q + 1 = 2^{h-1}$, следователно $\mathrm{bin}(q+1) = 10^{h-1}$. От друга страна, $p \geqslant q+1$, така че $\mathrm{bin}(p) = 1\sigma$ за някой булев стринг σ с дължина $h-1$. Тогава $\mathrm{bin}(p + q + 1) = 10\sigma$. С други думи, $\mathrm{bin}(n) = 10\sigma$. Но тогава $\mathrm{bin}_h(n) = 0\sigma = 0\mathrm{bin}_{h-1}(p)$.

  Доказахме, че наистина адресите на върховете от последното ниво на T са лексикографски сортираните стрингове от 0^h до $\mathrm{bin}_h(n)$. □

1.3 Двоична пирамида

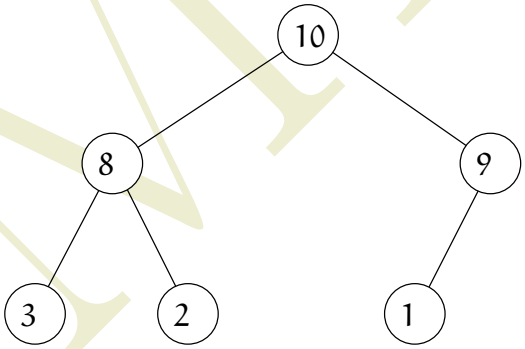
Сега допускаме, че всеки връх има асоциирана стойност, наречена *ключ* (key). Въз основа на "попълнено двоично дърво" въвеждаме "двоична пирамида" (binary heap, вж. [NIS]).

Определение 4 (двоична пирамида)**.** Двоична пирамида *е попълнено двоично дърво, в което следните факти са в сила за ключовете.*

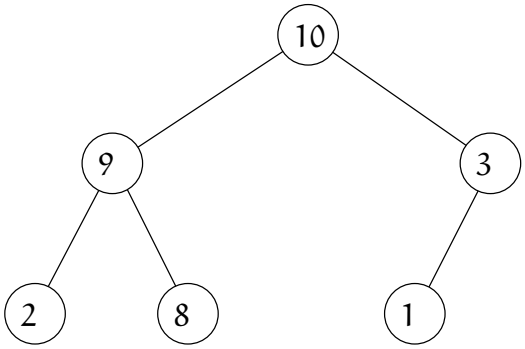
- *Ключът на всеки връх трябва да е по-голям или равен на ключовете на неговите деца. Тогава пирамидата е* максимална пирамида *(max heap).*
- *Ключът на всеки връх трябва да е по-малък или равен на ключовете на неговите деца. Тогава пирамидата е* минимална пирамида *(min heap).*

Тук ще разглеждаме максимални пирамиди и казвайки "пирамида" ще имаме предвид "максимална пирамида". Но изложението може да бъде променено лесно, така че да се отнася за минимални пирамиди. Също така, ще изпускаме "двоични", понеже не разглеждаме пирамиди, които не са двоични.

Ето пример за пирамида:



Очевидно фòрмата на дървото е фиксирана от броя на върховете, но подредбата на ключовете не е фиксирана[†]. Най-големият ключ 10 непременно е в корена и 9 непременно е в дете на корена, но известно вариране е възможно. Примерно, това също е пирамида със тези ключове:



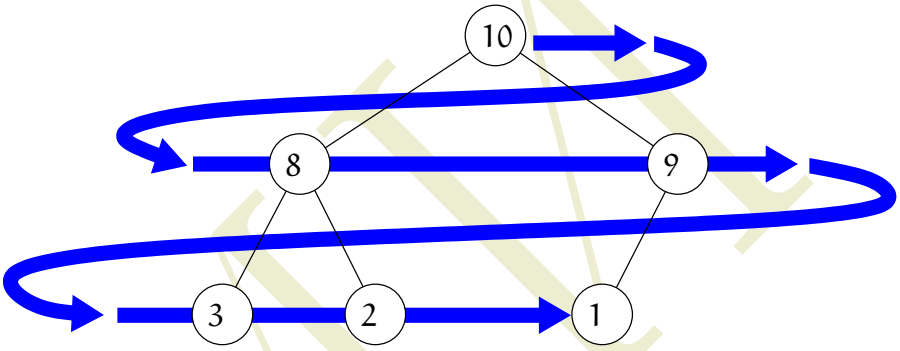
Наблюдение 1. *Попълнено двоично дърво с ключове е пирамида тогава и само тогава, когато за всяко листо* u*, по (уникалния) път* p*, свързващ корена с* u*, стойностите на ключовете са ненарастващи по* p *в посока от корена към* u*.* □

1.4 Реализиране на пирамида чрез масив

Определение 5. *Нека* $T = (V, E)$ *е попълнено двоично дърво с* n *върха и корен* r*.* Линеаризация *на* T *се нарича масивът* $A[1, 2, \ldots, n]$ *с елементи* V*, където за всяко* i*,* $1 \leqslant i \leqslant n$*,* $A[i]$ *е ключът на върха, чийто адрес е* i*.*

[†] В Секция 5 дискутираме броя на различните пирамиди с n върха

Съгласно Лема 6, съществува биекция между множеството от адресите и множеството $\{1, 2, \ldots, n\}$, така че определението е смислено. И така, пирамидите може да се реализират много ефикасно чрез масиви. Примерно, линеаризацията на последния пример е $[10, 8, 9, 3, 2, 1]$. Линеаризирането можем да си представяме като обхождане на дървото по нива, във всяко ниво, отляво надясно, и последователно записване на ключовете в масива:



Лема 6 гарантира, че в нивата няма "празнини", така че в масива няма "дупки".

При тази реализация се избягва разходът на памет, който е неизбежен при дърветата по принцип. Ако искаме да представим двоично дърво в общия случай—не непременно попълнено—трябва по някакъв начин да укажем за всеки връх кой е родителят, или кои са децата. Това ни струва допълнителна памет. Докато пирамида може да се реализира без никаква допълнителна памет, като при това обхождането е много ефикасно като сложност по време. Последното твърдение се основава на факта (виж Наблюдение 2), че ако масивът $A[1, \ldots, n]$ е линеаризация на пирамида, то за всеки елемент $A[i]$ можем в $\Theta(1)$ време да определим дали $A[i]$ е корен или не, ако не е, то кой е родителят му, дали е вътрешен връх или листо, и ако е вътрешен връх, кое е лявото му дете $A[j]$, ако такова съществува, и дясното му дете $A[k]$, ако такова съществува.

Следното наблюдение е просто приложение на Лема 5.

Наблюдение 2. *Нека T е пирамида с n елемента и масивът $A[1, \ldots, n]$ е нейната линеаризация. Тогава за всеки елемент $A[i]$:*

1. *елементът, който отговаря на родителя на $A[i]$, е $A\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$, ако върхът на пирамидата, който отговаря на $A[i]$, не е коренът.*
2. *елементът, който отговаря на лявото дете на $A[i]$, е $A[2 \times i]$, ако върхът от пирамидата, отговарящ на $A[i]$, има ляво дете,*
3. *елементът, който отговаря на дясното дете на $A[i]$, е $A[2 \times i + 1]$, ако върхът от пирамидата, отговарящ на $A[i]$, има дясно дете.* □

Наблюдение 2 ни дава основание да дефинираме следните изчислителни примитиви[†], чиито имена са доста-

[†]"Примитив" е, неформално казано, лесна за възприемане функция с малка сложност, която се използва често от алгоритмите, които ни интересуват.

тъчно информативни:

```
PARENT(i)
    return (if i ⩾ 2 then ⌊i/2⌋ else i)

LEFT(i)
    return (if 2i ⩽ n then 2i else UNDEFINED)

RIGHT(i)
    return (if 2i + 1 ⩽ n then 2i + 1 else UNDEFINED)

LEVEL(i)
    return ⌈log₂ i⌉

ISLEAF(i)
    return (if i > ⌊n/2⌋ then YES else NO)

ISINTERNALVERTEX(i)
    return not ISLEAF(i)

ISROOT(i)
    return (if i = 1 then YES else NO)
```

Освен това, за $t \geqslant 1$:

$$\text{PARENT}^t(i) = \begin{cases} \text{PARENT}(i), & \text{ако } t = 1 \\ \text{PARENT}\big(\text{PARENT}^{t-1}(i)\big), & \text{в противен случай} \end{cases}$$

Казваме, че $A[j]$ е *предшественик* на $A[i]$, ако $j = \text{PARENT}^t(i)$ за някое $t \geqslant 1$.

Определение 6 (инверсия в пирамида)**.** *Нека $A[1, \ldots, n]$ е масив от ключове.* Пирамидална инверсия *в $A[\,]$ наричаме всяка наредена двойка индекси $\langle j, i\rangle$, такава че $A[j]$ е предшественик на $A[i]$ и $A[j] < A[i]$. Наредената двойка $\langle A[j], A[i]\rangle$ също наричаме инверсия.* □

Примерно, на Фигура 8 е показан обект (като попълнено дърво, но читателят лесно може да си представи съответния масив), би бил пирамида, ако не беше последният елемент (най-дясното листо). Има три пирамидални инверсии, които са между последния елемент и негови предшественици.

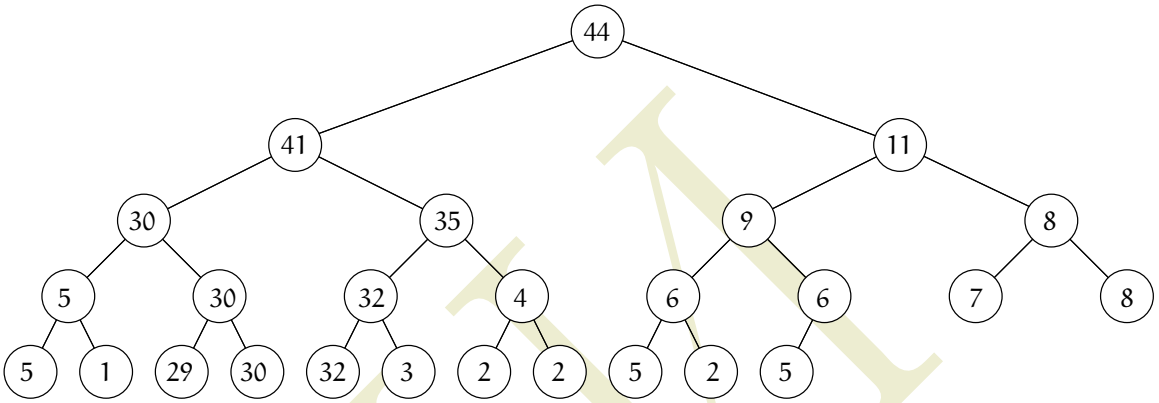
Наблюдение 3. *Масивът от ключове $A[\,]$ е пирамида тогава и само тогава, когато няма нито една пирамидална инверсия.* □

1.5 Подпирамида

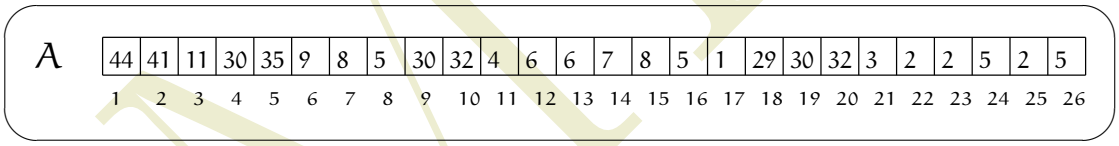
Нека T е пирамида с n върха, реализирана с масив $A[1, 2, \ldots, n]$. За всеки връх $u \in V(T)$, *подпирамидата с корен* u е $T[\![u]\!]$. По отношение на масива $A[\,]$, който преставя пирамидата, u е индекс на елемент. Нотацията $A[\![u]\!]$ означава реализацията на $T[\![u]\!]$ **в масива** $A[\,]$. Очевидно, в общия случай $A[\![u]\!]$ не е непрекъснат подмасив, а се състои от няколко подмасива—на брой колкото е височината на съответното поддърво—всеки от които е непрекъсната подпоследователност на $A[\,]$ (вижте Фигура 6).

Наблюдение 4. *Ако $A[1, \ldots, n]$ е пирамида, то за всяко $k \in \{1, \ldots, n\}$, $A[\![k]\!]$ е пирамида.* □

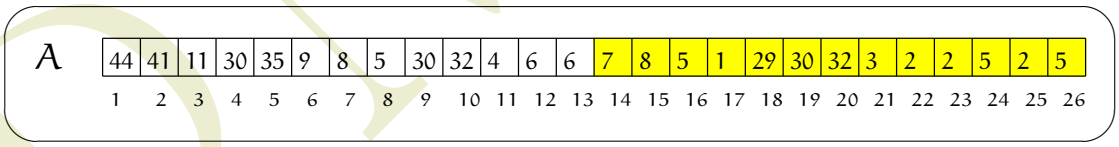
Като пример за пирамида и нейната линеаризация, да разгледаме пирамидата, показана на Фигура 3. Фигура 4 показва нейната линеаризация. Фигура 5 показва листата на пирамидата. На Фигура 6 е показана подпирамидата $A[\![3]\!]$. Връх 3 е третият връх в масива (дясното дете на корена, ако разсъждаваме в термините на дървото). Фигура 7 показва листата на $A[\![3]\!]$ в червено.



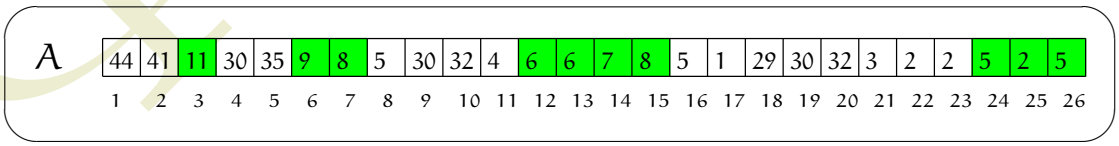
Фигура 3: Пирамида с 26 върха.



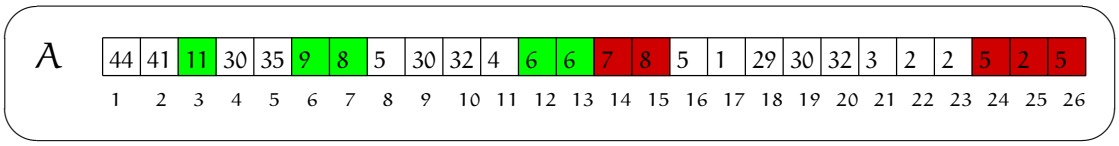
Фигура 4: Линеаризацията на пирамида от Фигура 3.



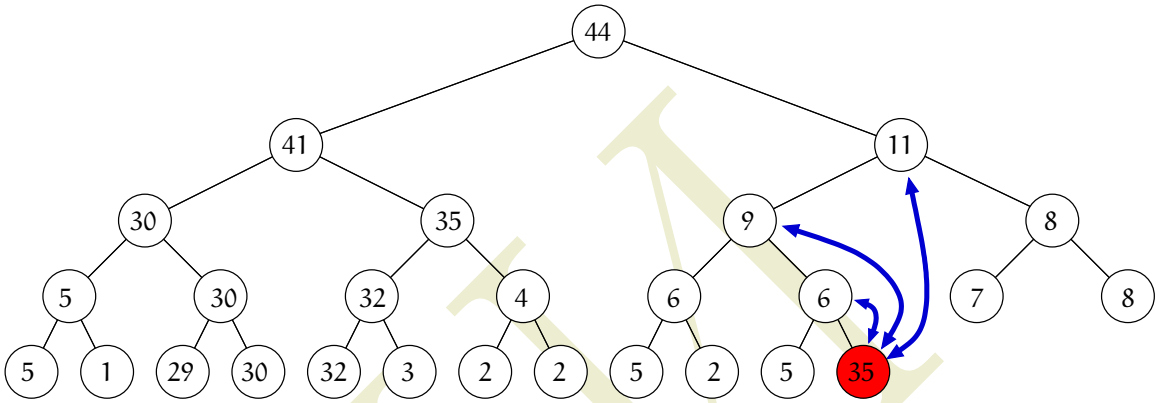
Фигура 5: Листата на пирамидата от Фигура 3 са отбелязани с жълто.



Фигура 6: Подпирамидата A⟦3⟧.



Фигура 7: Листата на A⟦3⟧ в червено.



Фигура 8: Пирамидата от Фигура 3 с един добавен връх (в червено). Полученият обект не е пирамида, понеже има три пирамидални инверсии между новодобавения елемент и негови предшественици. Пирамидалните инверсии са показани със сини стрелки.

Използвайки дефинираните примитиви, следният алгоритъм обхожда $T[\![u]\!]$. "Обхожда" означава "извежда ключовете на $T[\![u]\!]$ в *preorder*[†]" (а не по нива).

TRAVERSE BINARY SUBHEAP($A[1,\ldots,n]$: пирамида, i: число от $\{1,\ldots n\}$)
```
1  print i
2  left ← LEFT(i)
3  right ← RIGHT(i)
4  if left ⩽ n
5      TRAVERSE BINARY SUBHEAP(A[], left )
6  if right ⩽ n
7      TRAVERSE BINARY SUBHEAP(A[], right)
```

2 Построяване на пирамида

Даден е масив $A[1,\ldots,n]$ от ключове. За простота допускаме, че ключовете се естествени числа. Числата са в произволен порядък. Искаме да разместим числата така, че масивът да стане пирамида. Дори числата да са две по две различни, има много начини да направим пирамида от дадени числа. За подробна дискусия на броя на пирамидите вижте Секция 5.

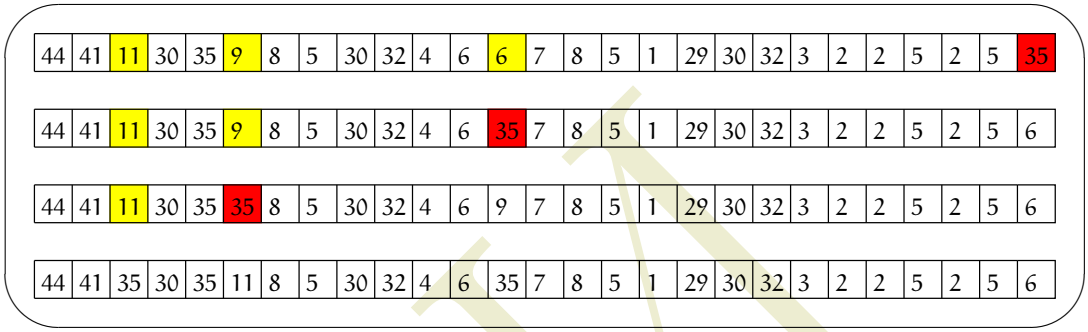
2.1 Наивно построяване на пирамида

Най-естественият начин да бъде превърнат произволен масив от числа в пирамида чрез разместване е чрез последователно добавяне на елементи в частично построена пирамида.

NAIVE BUILD HEAP, OUTLINE($A[1,\ldots,n]$: масив от ест. числа)
```
1  for i ← 2 to n
2      добави A[i] към досега построената пирамида A[1,...,i−1], така че A[1,...,i] да стане пирамида
```

Това описание е прекалено общо, но идеята е ясна: "добави към $A[1,\ldots,i-1]$" не е просто увеличаване на индекса i. Дори $A[1,\ldots,i-1]$ да е пирамида, $A[1,\ldots,i-1,i]$ не е непременно пирамида, защото може да има пирамидални инверсии $\langle \text{PARENT}^t(i), i\rangle$. Това е илюстрирано на Фигура 8, на която структурата е показана като дърво, а не като масив. Дървото съдържа пирамидата от Фигура 3 плюс още един връх, оцветен в червено. Ключът на този нов връх е 35 и той участва в пирамидални инверсии с три от своите предшественици. Съществуват много начини да бъдат разместени елементите на $A[1,\ldots,i]$, така че да няма пирамидални инверсии. Най-естественият и прост начин е, да ликвидираме пирамидалните инверсии, в които участва новодобавеният $A[i]$, без да разместваме никакви елементи, които не участват в тези инверсии. Ако мислим за обекта

[†]За обхождания на дървета, включително в preorder, вижте примерно [SW11].



Фигура 9: Линеаризацията на пирамидата от Фигура 3 е най-горе. Елементът, който "не си е на мястото", е накрая в червено. Елементите, които образуват пирамидални инверсии с него, са в жълто. Следващите три състояния на масива отгоре надолу илюстрират работата на Naive Build Heap.

като дърво: очевидно въпросните инверсии са между новодобавения $A[i]$ и върхове по пътя от него към корена, които индуцират подпът (тоест, по пътя от $A[i]$ към корена, върховете, участващи в инверсии, са непрекъсната последователност.) Очевидното решение тогава е да разменяме $A[i]$ с $A[\textsc{Parent}(i)]$, докато има инверсии. Всяка такава размяна ликвидира една инверсия и не въвежда нови инверсии, защото ключът, който застава "отгоре", е по-голям от този, който преди е бил там преди размяната, следователно този елемент (който застава отгоре) остава не по-малък от другия си наследник (ако има такъв).

Naive Build Heap, detailed($A[1,\ldots,n]$: масив от ест. числа)
```
1  for i ← 2 to n
2      while i > 1 and A[Parent(i)] < A[i]  do
3          swap(A[i], A[Parent(i)])
4          i ← Parent(i)
```

Ще илюстрираме работата на наивното построяване, използвайки за пример пирамидата от Фигура 3. Работата на алгоритъма е показана на Фигура 9.

Доказателството за коректност е напълно тривиално и остава за читателя. Ще изследваме сложността по време на наивното построяване на пирамида. Външният цикъл се изпълнява $\Theta(n)$ пъти без оглед на конкретния вход. Вътрешният цикъл се изпълнява, в най-лошия случай, $\lfloor \lg i \rfloor$ пъти за всяко i, понеже височината на подпирамидата с корен $A[i]$ е $\lfloor \lg i \rfloor$. Тогава сложността се определя от сумата

$$\sum_{i=2}^{n} \lfloor \lg i \rfloor = \Theta\left(\sum_{i=2}^{n} \lg i\right) = \Theta\left(\lg\left(2 \times 3 \times \ldots \times (n-1) \times n\right)\right) = \Theta\left(\lg n!\right) \tag{1}$$

Добре известно е (виж [CLRS09]), че $\lg n! \asymp n \lg n$. Следователно, сложността на наивното построяване е $\Theta(n \lg n)$.

2.2 Бързо построяване на пирамида: алгоритъм Build Heap

Алгоритъмът-предмет на тази подсекция е предложен от Robert Floyd [Flo64]. Самият алгоритъм е показан на стр. 22. Той използва функция Heapify, която показваме в два варианта: итеративен и рекурсивен.

2.2.1 Неформално въвеждане на Heapify

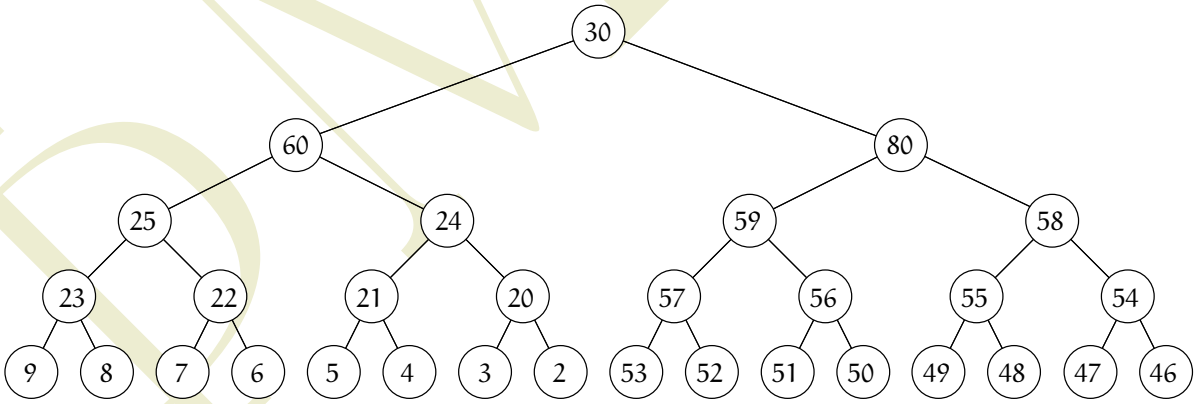
Да допуснем, че е дадено попълнено двоично дърво с ключове T. Нека коренът на T е u и децата на u са v и w. За целта на това обяснение, нека ключовете са съответно $u.key$, $v.key$ и $w.key$. Нека $T[\![v]\!]$ и $T[\![w]\!]$ са пирамиди. Ако $u.key \geqslant v.key$ и $u.key \geqslant w.key$, то T е пирамида. Да допуснем, че $u.key < v.key$ или $u.key < w.key$. Без ограничение на общността, нека $u.key < v.key$ и $u.key < w.key$. Тогава в T има много пирамидални инверсии. Пирамидалните инверсии може да са $n-1$ на брой, където n е броят на върховете на T: може всеки елемент от $T[\![v]\!]$ и $T[\![w]\!]$ да участва в инверсия с корена u.

Ключовото наблюдение е, че с $O(\lg n)$ размени на елементи може да премахнем всички пирамидални инверсии, тоест да направим T пирамида. Интуитивно е ясно, че трябва да разменим u с този връх измежду v и

w, чийто ключ е максимален[†]. Без ограничение на общността, нека $v.\text{key} < w.\text{key}$. Тогава ще разменим u с w. Със сигурност след размяната w няма да участва в инверсия с нито един връх от $T[\![v]\!]$, така че след размяната ще инверсии—ако изобщо има—само в поддървото, чийто корен сега е u. Грубо казано, броят на инверсиите ще намалее наполовина или повече.[‡]

Забележете, че има примери, в които е по-изгодно да разменим корена не с детето с по-големия ключ, а с детето с по-малкия ключ. Такова дърво е показано на Фигура 10. Първо да видим какво ще стане, ако върху това дърво прилагаме размяна на връх с това от двете деца, което има по-голям ключ. Фигура 11 показва дървото след първата размяна – елементите 30 и 80 са разменени, а с червено е очертан пътят, който ще “измине” върхът с ключ 30, движейки се надолу, докато не стане листо. На Фигура 12 е показано дървото след последната размяна, като е очертан пътят, който е “изминал” върхът с ключ 30. Това дърво е пирамида, получена след четири размени. За дървото от Фигура 10, ако разменим корена с детето с *по-малък* ключ, повече размени в лявото поддърво няма да има; ако разменим след това новия корен с ключ 60 с дясното дете с ключ 80, ще получим пирамидата, показана на Фигура 13, със само две размени.

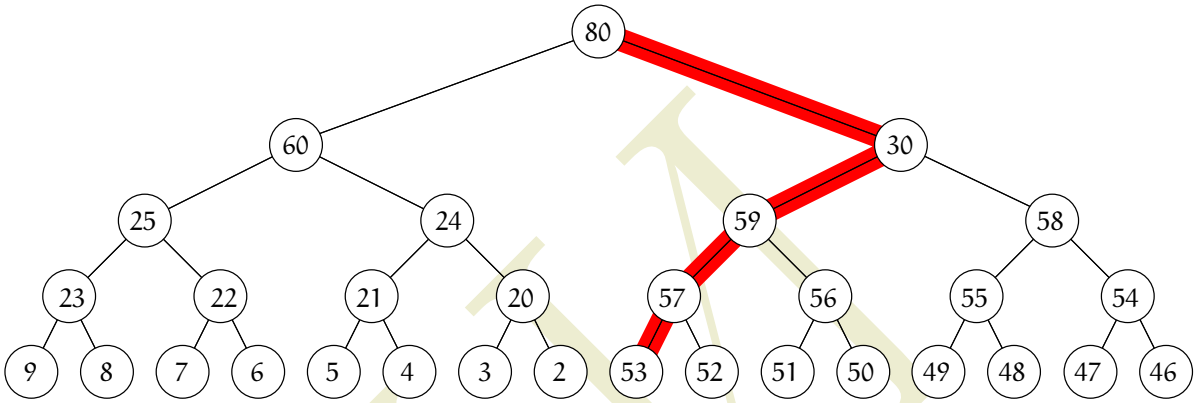
Примерът, който видяхме току-що, се мащабира за всякакъв размер на дървото, така че наистина има случаи, в които размяната с детето с по-малък ключ води до $\Theta(1)$ размени, а размяната с детето с по-голям ключ води до $\Theta(\lg n)$ размени. Но ние се интересуваме от сложността в най-лошия случай, а в най-лошия случай, размяната с детето с по-малък ключ може да доведе до $\Omega(n)$ размени, преди да получим пирамида, докато размяната с детето с по-голям ключ води до $O(\lg n)$ размени винаги. Пример за дърво, върху което размяната с детето с по-малък ключ би довела до $\Omega(n)$ размени (примерът се мащабира за безброй много n) е показан на Фигура 14.



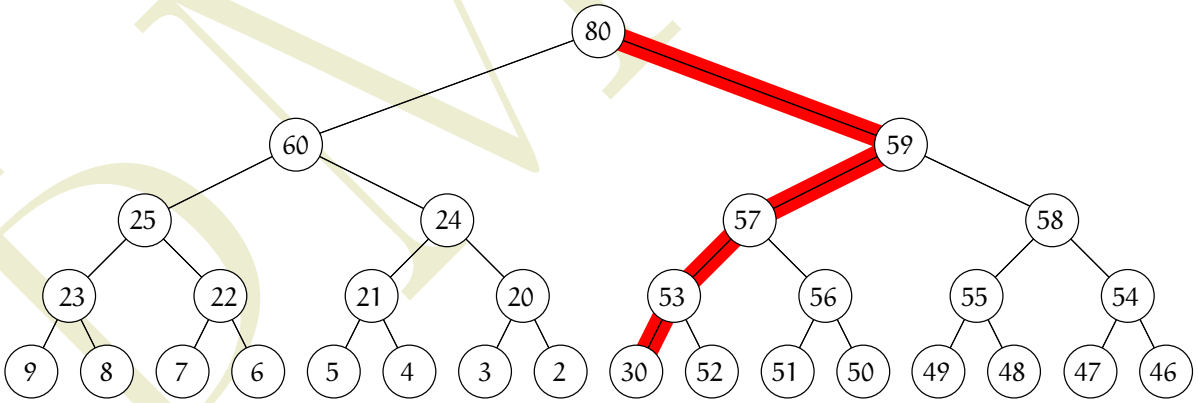
Фигура 10: Попълнено дърво с ключове, в което двете деца на корена са корени на поддървета-пирамиди, а ключът на корена е по-малък от ключовете на двете му деца. По-изгодно като минимален брой размени е коренът да бъде разменен с детето с по-малкия ключ.

[†]Ако v и w имат еднакви ключове, ще разменим u с кой да е от тях.

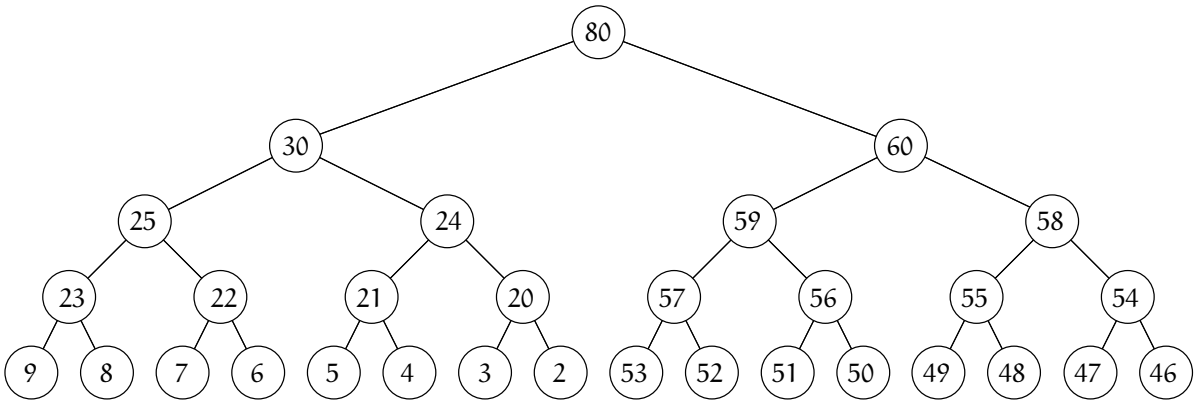
[‡]Лесно е да се съобрази, че максималният брой инверсии след размяната, изразен чрез общия брой на върховете, е $\frac{2(n-2)}{3}$, където $n-2$ се дели на 3. Дори броят на инверсиите да намалява след всяка размяна с делене на $\frac{3}{2}$, броят на размените е логаритмичен.



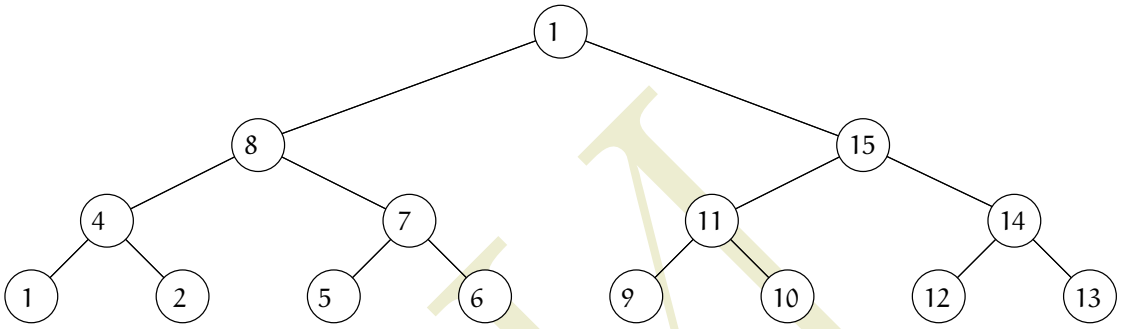
Фигура 11: По отношение на дървото от Фигура 10, ако разменим корена с детето с по-голям ключ и продължим размените по същия начин—с детето с по-голям ключ—ще направим 4 размени, докато върхът с ключ 30 не стане листо. Тук е показано дървото след първата размяна и е очертан пътят, който "изминава" върхът с ключ 30.



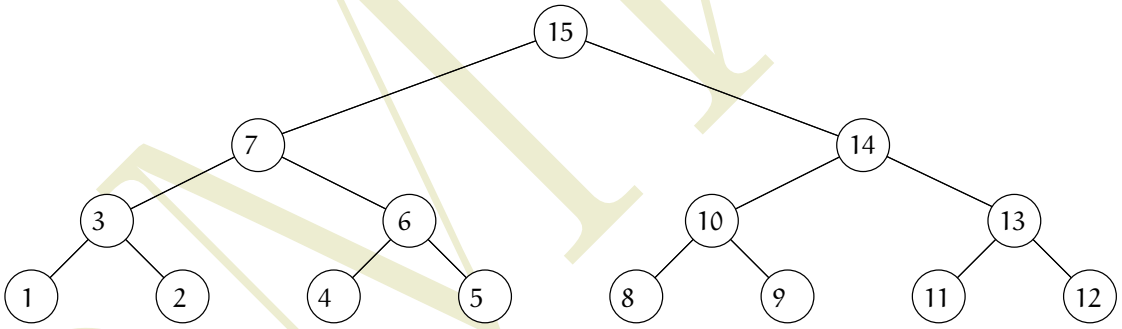
Фигура 12: По отношение на дървото от Фигура 11 е показано дървото след всички размени, където размените са от вида "разменяме корен с това дете, което има по-голям ключ". Извършени са 4 размени и дървото е пирамида. Очертан е пътят, който е "изминал" върхът с ключ 30.



Фигура 13: По отношение на дървото от Фигура 10, ако разменим корена с детето с **по-малък** ключ и продължим размените по същия начин—с детето с **по-малък** ключ—ще направим само 2 размени, и ще получим показаната пирамида.



Фигура 14: Пример за дърво, в което, ако разменяме корена с детето с по-малък ключ (и после правим още една размяна на новия корен с детето с по-голям ключ), броят на размените е по-голям от броя на върховете.



Фигура 15: Ако в дървото, показано на Фигура 14, систематично разменяме първо корен с детето с по-малък ключ, и после новия корен с детето с по-голям ключ, ще получим пирамидата, показана тук. Броят на размените е по-голям от броя на върховете.

От изложението трябва да е станало ясно, че предложената идея за Heapify би имала сложност $O(\lg n)$, ако се реализира грамотно: с $O(\lg n)$ размени на елементи унищожаваме всички пирамидални инверсии, а всяка размяна става в константно време.

2.2.2 Heapify в итеративен вариант

Iterative Heapify($A[1,2,\ldots,n]$: масив от цели числа, i: индекс в $A[\,]$)

```
1   j ← i
2   while j ⩽ ⌊n/2⌋ do
3       left ← Left(j)
4       right ← Right(j)
5       if left ⩽ n and A[left] > A[j]
6           largest ← left
7       else
8           largest ← j
9       if right ⩽ n and A[right] > A[largest]
10          largest ← right
11      if largest ≠ j
12          swap(A[j], A[largest])
13          j ← largest
14      else
15          break
```

Лема 7. *При допускането, че $A[1,\ldots,n]$ и i са такива, че всеки от $A[\![\text{Left}(i)]\!]$ и $A[\![\text{Right}(i)]\!]$, ако съществува, е пирамида, ефектът от* Iterative Heapify *е, че $A[\![i]\!]$ е пирамида при неговото приключване.*

Доказателство:
Това е инварианта за **while**-цикъла (редове 1–15):

> Всеки път, когато изпълнението е на ред 2, единствените възможни пирамидални инверсии в $A[\![i]\!]$ са $\langle j, \textsc{Left}(j)\rangle$ и $\langle j, \textsc{Right}(j)\rangle$, ако $A[\textsc{Left}(j)]$ и $A[\textsc{Right}(j)]$ съществуват.

База. $j = i$. Съгласно допусканията, всеки от $A[\![\textsc{Left}(i)]\!]$ и $A[\![\textsc{Right}(i)]\!]$, ако съществува, е пирамида, така че твърдението е в сила. ✓

Поддръжка. Допускаме, че твърдението е в сила в някой момент, в който изпълнението е на ред 2 и изпълнението ще бъде на ред 2 поне още веднъж.[†] И така, $j \leqslant \lfloor \frac{n}{2} \rfloor$ и поне едно от $\textsc{Left}(j) \leqslant n$ и $\textsc{Right}(j) \leqslant n$ е изпълнено, а именно $\textsc{Left}(j) \leqslant n$. Тогава поне $A[\textsc{Left}(j)]$ е дефинирано. Без ограничение на общността ще допуснем, че и $\textsc{Left}(j) \leqslant n$, и $\textsc{Right}(j) \leqslant n$, за да избегнем ненужни подслучаи.

Случай I: $A[\mathit{left}] > A[j]$ и $A[\mathit{right}] > A[j]$ в началото на изпълнението на цикъла, и освен това $A[\mathit{right}] > A[\mathit{left}]$. Условието на ред 5 е Истина и присвояването на ред 6 се случва. Условието на ред 9 също е Истина, така че и присвояването на ред 10 се случва. Когато изпълнението е на ред 11, $\mathit{largest}$ е равно на right. Условието на ред 11 е Истина и на ред 12, $A[j]$ и $A[\mathit{largest}]$ биват разменени. Твърдим, че единствените възможни пирамидални инверсии в $A[\![i]\!]$ след размяната са $\langle \mathit{right}, \textsc{Left}(\mathit{right})\rangle$ и $\langle \mathit{right}, \textsc{Right}(\mathit{right})\rangle$:

- нито един елемент от $A[\![i]\!]$, който е извън $A[\![j]\!]$, не е променян;
- $A[j] > A[\mathit{left}]$, така че $\langle j, \textsc{Left}(j)\rangle$ не може да е пирамидална инверсия;
- $A[j] > A[\mathit{right}]$, така че $\langle j, \textsc{Right}(j)\rangle$ не може да е пирамидална инверсия;
- $A[\![\mathit{left}]\!]$ не е бил модифициран;
- нито един от $A[\![\textsc{Left}(\mathit{right})]\!]$ и $A[\![\textsc{Right}(\mathit{right})]\!]$ не е бил модифициран.

Но на ред 13, j става равен на right. Следователно, инвариантата е в сила при следващото достигане на ред 2.

Случай II: $A[\mathit{left}] > A[j]$ и $A[\mathit{right}] > A[j]$ в началото на изпълнението на цикъла, но $A[\mathit{right}] \not> A[\mathit{left}]$. Условието на ред 5 е Истина и присвояването на ред 6 се случва. Условието на ред 9 е Лъжа, следователно присвояването на ред 10 не се случва и когато изпълнението е на ред 11, $\mathit{largest}$ е равно на left. Условието на ред 11 е Истина и на ред 12, $A[j]$ и $A[\mathit{largest}]$ биват разменени. Твърдим, че единствените възможни пирамидални инверсии в $A[\![i]\!]$ след размяната са $\langle \mathit{left}, \textsc{Left}(\mathit{left})\rangle$ и $\langle \mathit{left}, \textsc{Right}(\mathit{left})\rangle$:

- нито един елемент от $A[\![i]\!]$, който е извън $A[\![j]\!]$, не е променян;
- $A[j] > A[\mathit{left}]$, така че $\langle j, \textsc{Left}(j)\rangle$ не може да е пирамидална инверсия;
- $A[j] \geqslant A[\mathit{right}]$, така че $\langle j, \textsc{Right}(j)\rangle$ не може да е пирамидална инверсия;
- $A[\![\mathit{right}]\!]$ не е бил модифициран;
- нито един от $A[\![\textsc{Left}(\mathit{left})]\!]$ и $A[\![\textsc{Right}(\mathit{left})]\!]$ не е бил модифициран.

Но на ред 13, j става равен на left. Следователно, инвариантата е в сила при следващото достигане на ред 2.

Случай iii: $A[\mathit{left}] \not> A[j]$ и $A[\mathit{right}] > A[j]$ в началото на изпълнението на цикъла. Условието на ред 5 е Лъжа пристояването на ред 8 се случва. Условието на ред 9 е True, така че присвояването на ред 10 се случва и когато изпълнението е на ред 11, $\mathit{largest}$ е равно на right. Условието на ред 11 е Истина и на ред 12, $A[j]$ и $A[\mathit{largest}]$ биват разменени. Твърдим, че единственото възможни пирамидални инверсии в $A[\![i]\!]$ след размяната са $\langle \mathit{right}, \textsc{Left}(\mathit{right})\rangle$ и $\langle \mathit{right}, \textsc{Right}(\mathit{right})\rangle$. Доказателството е точно като доказателството на аналогичното твърдение в **Случай I**. Но на ред 13, j става равен на $\mathit{largest}$. Следователно, инвариантата е в сила при следващото достигане на ред 2.

Случай IV: $A[\mathit{left}] > A[j]$ и $A[\mathit{right}] \not> A[j]$ в началото на изпълнението на цикъла. Условието на ред 5 е True, така че присвояването на ред 6 се случва. Условието на ред 9 е False, така че присвояването на ред 10 не се случва и когато изпълнението е на ред 11, $\mathit{largest}$ е равно на left. Условието на ред 11 е True и на ред 12, $A[j]$ и $A[\mathit{largest}]$ биват разменени. Твърдим, че единствените в $A[\![i]\!]$ след размяната са $\langle \mathit{left}, \textsc{Left}(\mathit{left})\rangle$ и

[†] Последното имплицира, че ред 15 няма да бъде изпълнен.

$\langle \mathit{left}, \textsc{Right}(\mathit{left})\rangle$. Доказателството е точно като доказателството на аналогичното твърдение в **Случай II**. Но на ред 13, j става равен на $\mathit{largest}$. Следователно, инвариантата е в сила при следващото достигане на ред 2.

Случай V: $A[\mathit{left}] \not> A[j]$ и $A[\mathit{right}] \not> A[j]$ в началото на изпълнението на цикъла. Но това е невъзможно при текущите допускания, защото очевидно тогава ред 15 би бил достигнат и изпълнението не би се върнало повече на ред 2.

Терминация. Цикълът може да бъде напуснат по два начина: през ред 2, когато $j > \lfloor \frac{n}{2} \rfloor$, и през ред 15. Да разгледаме първата възможност. Тогава и $\textsc{Left}(j)$, и $\textsc{Right}(j)$ са индекси извън $A[1,\ldots,n]$. В този случай инвариантата казва, че няма пирамидални инверсии в $A[\![i]\!]$ изобщо, понеже $A[\textsc{Left}(j)]$ и $A[\textsc{Right}(j)]$ не съществуват. Следователно, $A[\![i]\!]$ е пирамида.

Да разгледаме втората възможност, а именно **while**-цикълът да бъде напуснат през ред 15. Очевидно, за да бъде достигнат ред 15, трябва да бъде вярно, че $\mathit{largest} = j$ на ред 11. А за да е вярно това, и $A[\mathit{left}] \leqslant A[j]$, и $A[\mathit{right}] \leqslant A[j]$ трябва да са в сила в началото на изпълнението на цикъла, понеже това е единственият начин ред 8 да бъде достигнат и ред 10 да не бъде достигнат. Но ако $A[\mathit{left}] \leqslant A[j]$ и $A[\mathit{right}] \leqslant A[j]$, пирамидални инверсии в $A[\![i]\!]$ няма, така че $A[\![i]\!]$ е пирамида. □

2.2.3 Heapify в рекурсивен вариант

```
RECURSIVE HEAPIFY(A[1,2,...,n]: масив от цели числа, i: индекс в A[])
 1  left ← LEFT(i)
 2  right ← RIGHT(i)
 3  if left ⩽ n  and A[left] > A[i]
 4      largest ← left
 5  else
 6      largest ← i
 7  if right ⩽ n  and A[right] > A[largest]
 8      largest ← right
 9  if largest ≠ i
10      swap(A[i], A[largest])
11      RECURSIVE HEAPIFY(A[], largest)
```

Лема 8. *При допускането, че* $A[1,\ldots,n]$ *и* i *са такива, че всеки от* $A[\![\textsc{Left}(i)]\!]$ *и* $A[\![\textsc{Right}(i)]\!]$, *ако съществува, е пирамида, ефектът от* Iterative Heapify *е, че* $A[\![i]\!]$ *е пирамида при неговото приключване.*

Доказателство:
По индукция по височината h на $A[\![i]\!]$.

> *Преди да направим доказателството, ще подчертаем следното: допусканията за* $A[\![\textsc{Left}(i)]\!]$ *и* $A[\![\textsc{Right}(i)]\!]$*—а именно, че са пирамиди—са* нещо напълно различно *от индуктивното предположение. Индуктивното предположение е твърдението, което доказваме, формулирано чрез някаква стойност на параметъра, на която сме дали име. Индуктивното предположение не е част от условията на лемата, докато споменатите допускания са част от условията на лемата.*

База. $h = 0$. Тогава $A[\![i]\!]$ съдържа един единствен елемент $A[]$. С други думи, това е листо. Тогава и $\textsc{Left}(i)$, и $\textsc{Right}(i)$ са индекси извън $A[]$. Да проследим изпълнението на Recursive Heapify: условието на ред 3 е Лъжа, следователно присвояването на ред 6 се случва. Условието на ред 7 също е Лъжа, така че ред 8 не се изпълнява и изпълнението преминава към ред 9. Условието там е Лъжа и текущото рекурсивно извикване терминира. Очевидно $A[\![i]\!]$ е пирамида при терминирането. ✓

Индуктивно предположение. Да допуснем, че за всяко $A[\![j]\!]$ от височина $\leqslant h-1$ с корен някой j, такъв че $A[j]$ е в $A[\![i]\!]$, е изпълнено, че Recursive Heapify$(A[], j)$ превръща $A[\![j]\!]$ чрез разместявания в пирамида.

Индуктивна стъпка. Да разгледаме изпълнението на Recursive Heapify$(A[], i)$. Без ограничение на общността, да допуснем, че $\textsc{Left}(i) \leqslant n$ и $\textsc{Right}(i) \leqslant n$, така че редове 3 и 7 са съответно

if $A[\mathit{left}] > A[i]$

и

if $A[right] > A[largest]$

Случай I: $A[left] > A[i]$, $A[right] > A[i]$ и $A[right] > A[left]$. Условието на ред 3 е Истина и присвояването на ред 4 се случва. Условието на ред 7 също е Истина, така че и присвояването на ред 8 се случва. Когато изпълнението е на ред 9, $largest$ е равно на $right$. Условието на ред 9 е Истина и на ред 10, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното повикване на ред 11 се случва, като бившият $A[i]$ сега е на позиция $right$. Съгласно индуктивното предположение, това рекурсивно викане построява пирамида от от $A[\![right]\!]$. От друга страна, $A[\![left]\!]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[right]$, е:

- по-голям от текущия $A[left]$ по допускане;
- не по-малък от текущия $A[right]$, по следните причини. Първоначално, $A[\![right]\!]$ е пирамида, така че бившият $A[right]$ е не по-малък от всеки друг елемент от $A[\![right]\!]$ в онзи момент (в началото). В края, елементите на $A[\![right]\!]$ се състоят от началните елементи без началния $A[right]$, плюс началния $A[i]$. Тъй като началният $A[right]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\![right]\!]$, текущият $A[i]$ не е по-малък от текущия $A[right]$.

Тогава, $A[\![i]\!]$ е пирамида.

Случай II: $A[left] > A[i]$, $A[right] > A[i]$ и $A[right] \not> A[left]$. Условието на ред 3 е Истина и присвояването на ред 4 се случва. Условието на ред 7 е Лъжа, така че присвояването на ред 8 не се случва и когато изпълнението е на ред 9, $largest$ е равно на $left$. Условието на ред 9 е Истина и на ред 10, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното извикване на ред 11 се случва, с бившия $A[i]$ сега на позиция $left$. Съгласно индуктивното предположение, това рекурсивно извикване построява пирамида от $A[\![left]\!]$. От друга страна, $A[\![right]\!]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[left]$, е:

- не по-малък от текущия $A[right]$ по допускане
- не по-малък от текущия $A[left]$, по следните причини. Първоначално, $A[\![left]\!]$ е пирамида, така че бившият $A[left]$ е не по-малък от всеки друг елемент на $A[\![left]\!]$ в онзи момент (в началото). В края, елементите на $A[\![left]\!]$ се състоят от началните елементи без началния $A[left]$, плюс началния $A[i]$. Тъй като началният $A[left]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\![left]\!]$, текущият $A[i]$ не е по-малък от текущия $A[left]$.

Тогава, $A[\![i]\!]$ е пирамида.

Случай III: $A[left] \not> A[i]$ и $A[right] > A[i]$. Условието на ред 3 е Лъжа и присвояването на ред 6 се случва. Условието на ред 7 е Истина, така че присвояването на ред 8 се случва и когато изпълнението е на ред 9, $largest$ е равно на $right$. Условието на ред 9 е Истина и на ред 10, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното извикване на ред 11 се случва, с бившия $A[i]$ сега на позиция $right$. Съгласно индуктивното предположение, това рекурсивно извикване построява пирамида от $A[\![right]\!]$. От друга страна, $A[\![left]\!]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[right]$, е:

- по-голям от текущия $A[left]$ заради допусканията и транзитивността на неравенствата;
- не по-малък от текущия $A[right]$ по следните причини. Първоначално, $A[\![right]\!]$ е пирамида, така че бившият $A[right]$ е не по-малък от всеки друг елемент на $A[\![right]\!]$ в онзи момент (в началото). В края, елементите на $A[\![right]\!]$ се състоят от началните елементи без началния $A[right]$, плюс началния $A[i]$. Тъй като началният $A[right]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\![right]\!]$, текущият $A[i]$ не е по-малък от текущия $A[right]$.

Тогава, $A[\![i]\!]$ е пирамида.

Случай IV: $A[left] > A[i]$ и $A[right] \not> A[i]$. Условието на ред 3 е Истина и присвояването на ред 4 се случва. Условието на ред 7 е Лъжа, така че присвояването на ред 8 не се случва и когато изпълнението е на ред 9, $largest$ е равно $left$. Условието на ред 9 е Истина и на ред 10, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното извикване на ред 11 се случва, с бившия $A[i]$ сега на позиция $left$. Съгласно индуктивното предположение, това рекурсивно извикване построява пирамида от $A[\![left]\!]$. От друга страна, $A[\![right]\!]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[left]$, е:

- по-голям от текущия $A[right]$ заради допусканията и транзитивността на неравенствата;
- не по-малък от текущия $A[left]$ поради следните причини. Първоначално, $A[\![left]\!]$ е пирамида, така че бившият $A[left]$ е не по-малък от всеки друг елемент от $A[\![left]\!]$ в онзи момент (в началото). В края, елементите на $A[\![left]\!]$ се състоят от началните елементи без началния $A[left]$, плюс началния $A[i]$. Тъй като началният $A[left]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\![left]\!]$, текущият $A[i]$ не е по-малък от текущия $A[left]$.

Тогава, $A[\![i]\!]$ е пирамида.

Случай V: $A[left] \not> A[i]$ и $A[right] \not> A[i]$. Условието на ред 3 е Лъжа и присвояването на ред 6 се случва. Условието на ред 7 също е Лъжа, така че присвояването на ред 8 не се случва и когато изпълнението достигне ред 9, $largest$ е равно на i. Условието на ред 9 е Лъжа и изпълнението терминира, оставяйки $A[\![i]\!]$ непроменен. Съгласно допусканията, $A[\![left]\!]$ и $A[\![right]\!]$ са пирамиди и $A[left] \not> A[i]$ и $A[right] \not> A[i]$. Тогава, $A[\![i]\!]$ е пирамида. □

2.2.4 Алгоритъм Build Heap

Идеята на бързия алгоритъм за построяване на пирамида е да сканира масива отдясно наляво, иначе казано, от листата към корена, започвайки от първото не-листо. Листата са тривиални пирамиди.

За всяко не-листо $A[i]$ в указания ред, при допускането, че и двете му деца (или едното му дете, ако е само едно) са корени на подпирамиди, ако $A[i]$ е по-голям или равен от двете си деца (или едното си дете), то $A[\![i]\!]$ е подпирамида. Ако това не е вярно, ще направим $A[\![i]\!]$ пирамида чрез серия от размени на елементи в нея, разменяйки корен с по-голямото дете, избутвайки малкия елемент, който образува пирамидални инверсии, в посока към листата, докато той или не стане листо, или не се окаже по-голям или равен на децата си (детето си).

Това всъщност означава, че викаме Heapify върху всяко нелисто, отдясно наляво. Функцията Heapify в следния алгоритъм е или Recursive Heapify на стр. 20, или Iterative Heapify на стр. 18.

```
Build Heap(A[1,2,...,n]: array of integers)
1   for i ← ⌊n/2⌋ downto 1
2       Heapify(A[ ], i)
```

Лема 9. *За всеки вход* $A[\,]$, Build Heap *построява пирамида от него.*

Доказателство:
Следното твърдение е инварианта на **for**-цикъла (редове 1–2):

> Всеки път, когато изпълнението е на ред 1, $A[\![i+1]\!]$, $A[\![i+2]\!]$, ..., $A[\![n]\!]$ са пирамиди.

База. При първото достигане на ред 1, i е равно на $\lfloor \frac{n}{2} \rfloor$. Тогава $A[\![\lfloor \frac{n}{2} \rfloor + 1]\!]$, $A[\![\lfloor \frac{n}{2} \rfloor + 2]\!]$, ..., $A[\![n]\!]$ са точно листата на попълненото дърво, чиято линеаризация е $A[\,]$. Очевидно всяко листо е пирамида. ✓

Поддръжка. Да разгледаме някое достигане на ред 1, което не е последното. И $A\langle \textsc{Left}(i)]\!]$, и $A\langle \textsc{Right}(i)]\!]$ (ако съществува) са пирамиди съгласно индуктивното предположение, понеже $i < \textsc{Left}(i)$ и $i < \textsc{Right}(i)$. Тогава използваме доказаната коректност на Heapify и заключаваме, че $A[\![i]\!]$ става пирамида. След това i намалява с единица. Спрямо новата стойност на i, инвариантата е в сила. ✓

Терминация. Когато изпълнението е на ред 2 за последден път, $i = 0$. Тогава $A[\![0+1]\!]$, $A[\![0+2]\!]$, ..., $A[\![n]\!]$ са пирамиди. В частност, $A[\![1]\!]$ е пирамида. Но $A[\![1]\!]$ е $A[\,]$. Това доказва лемата. ✓ □

Лема 10. Build Heap *работи във време* $\Theta(n)$.

Доказателство:
Това, че сложността по време е $\Omega(n)$, е очевидно. Ще покажем, че сложността е $O(n)$. Асимптотична горна граница за сложността на Build Heap е сумата по всички височини от 1 (Build Heap прескача листата) до $\lfloor \lg n \rfloor$ (височината на дървото) от произведението от броя на върховете с дадената височина и самата височина (всеки връх може да се "придвижва" надолу към листата с най-много толкова размени, колкото е височината

му поначало). Съгласно Следствие 4, броят на върховете с височина k е $\left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil$. И така, ако $T(n)$ е функцията на сложността, то в най-лошия случай,

$$T(n) = \sum_{k=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil \Theta(k) = \sum_{k=1}^{\lfloor \lg n \rfloor} \Theta\left(\frac{n}{2^k} k\right)$$

Да разгледаме $\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k$. В сила е

$$\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k \leqslant \sum_{k=1}^{\infty} \frac{n}{2^k} k = n \sum_{k=1}^{\infty} \frac{k}{2^k}$$

Тривиално се доказва, че редът $\sum_{k=1}^{\infty} \frac{k}{2^k}$ е сходящ, примерно с критерия на d'Alembert:

$$\lim_{k \to \infty} \frac{\frac{k+1}{2^{k+1}}}{\frac{k}{2^k}} = \frac{1}{2} < 1$$

Щом $\sum_{k=1}^{\infty} \frac{k}{2^k}$ е ограничен от константа, то $n \sum_{k=1}^{\infty} \frac{k}{2^k} \asymp n$ и очевидно $\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k \asymp n$. Тогава $T(n) \asymp n$. □

3 Сортиращ алгоритъм Heapsort

Следният сортиращ алгоритъм е предложен от Williams [Wil64]. Той е първият от бързите сортиращи алгоритми, които ще разгледаме. "Бърз" означава, сложност по време $O(n \lg n)$.

Нека A.size относно масива $A[1, \ldots, n]$ е число, такова че $1 \leqslant A.size \leqslant n$ и Heapify работи върху подмасива $A[1, \ldots, A.size]$, а **не** върху $A[1, \ldots, n]$.

```
Heapsort(A[1,2,...,n])
1   Build Heap(A[])
2   A.size ← n
3   for i ← n downto 2
4       swap(A[1], A[i])
5       A.size ← A.size − 1
6       Heapify(A[], 1)
```

Лема 11. Heapsort *е сортиращ алгоритъм.*

Доказателство:
Нека $A'[1, \ldots, n]$ означава първоначалния масив. Следното твърдение е инварианта на цикъла за **for**-цикъла (редове 3–6):

> Всеки път, когато изпълнението на Heapsort е на ред 3:
>
> - Текущият подмасив $A[i+1, \ldots, n]$ се състои от $n-i$ на брой най големи елементи на $A'[1, \ldots, n]$ в сортиран вид.
> - Освен това, текущият $A[1, \ldots, i]$ е пирамида.

База. При първото достигане на ред 3, $i = n$. Подмасивът $A[i+1, \ldots, n]$ е празен, следователно, в празния смисъл (*vacuously*), той се състои от нула на брой най-големи елементи от $A'[1, \ldots, n]$, в сортиран вид, следователно първата част на инвариантата е в сила. $A[1, \ldots, n]$ е пирамида съгласно Лема 11, приложена към ред 1, така че и втората част на инвариантата е в сила. ✓

Поддръжка. Да допуснем, че твърдението е в сила при някакво достигане на ред 3, което не е последното. Нека да наричаме масива $A[\,]$ в този момент, $A''[\,]$. Съгласно първата част на индуктивното предположение, $A''[i+1, \ldots, n]$ се състои от $n-i$ на брой елементи от $A'[\,]$ в сортиран вид. Съгласно втората част на индуктивното предположение, $A''[1]$ е максимален елемент от $A''[1, \ldots, i]$. След размяната на ред 4, $A''[i, \ldots, n]$ $n-i+1$ на брой най-големи елемента на $A'[\,]$ в сортиран вид. Спрямо новата стойност на i при следващото достигане на ред 3, първата част на инвариантата е в сила.

Ще докажем, че втората част на инвариантата е в сила. Да приложим Лема 7 или Лема 8 в зависимост от това дали ползваме рекурсивна или итеративна HEAPIFY функция на ред 6. Трябва да се има предвид, че HEAPIFY работи върху $A[1, \ldots, i-1]$, защото i е равно на A.size в момента , в който изпълнението е на ред 6, а на ред 5, A.size е получил стойност $i-1$. Заради това, на ред 6 текущият $A[i]$ е "извън обхвата" на HEAPIFY.

Терминация. Да разгледаме момента, в който изпълнението е на ред 3 за последен път. Очевидно, $i = 1$. Да заместим 1 на мястото на i в инвариантата. Получаваме "текущият подмасив $A[2, \ldots, n]$ се състои от $n-1$ на брой най-големи елементи на $A'[1, \ldots, n]$ в сортиран вид". Но тогава $A[1]$ е минимален елемент на $A'[1, \ldots, n]$. С това доказателството за коректност на HEAPSORT приключва. □

Да разгледаме сложността по време на HEAPSORT. Сложността на BUILD HEAP е $\Theta(n)$, което доказахме в Лема 10. Сложността на **for**-цикъла (редове 3–6) $\Theta(n \lg n)$, защото за всяко i, сложността на HEAPIFY в най-лошия случай е $\Theta(\lg i)$, а ние вече изследвахме сумата $\sum_{i=1}^{n} \lg i$ (вж. (1)) и установихме, че асимптотично ѝ нарастване е $\Theta(n \lg n)$. Следователно, сложността на HEAPSORT е $\Theta(n) + \Theta(n \lg n) = \Theta(n \lg n)$.

Сложността по памет на HEAPSORT е $\Theta(1)$.

HEAPSORT не е стабилен сортиращ алгоритъм. За да се убедим в това, достатъчно е да разгледаме работата му над вход от еднакви ключове. При дадения псевдокод, BUILD HEAP няма да промени нищо, след което още първото изпълнение на **for**-цикъла ще размени еднаквите елементи $A[1]$ и $A[n]$, като този елемент, който бива записан в $A[n]$, ще остане там до края на алгоритъма.

Дори да сложим проверка на ред 4, такава размяната да не се изпълнява, ако $A[1]$ и $A[i]$ са равни, това няма да направи алгоритъма стабилен. Да си представим вход от k ключа 2 и k ключа 1, като двойките са преди единиците. BUILD HEAP няма да промени нищо, след което още първото изпълнение на **for**-цикъла ще размени двойката в $A[1]$ с единицата в $A[n]$, при което двойката, която е била вляво от всички други двойки, ще се окаже вдясно от тях, и ще остане така до края на алгоритъма.

4 Приоритетни опашки

4.1 Абстрактен Тип Данни (АТД)

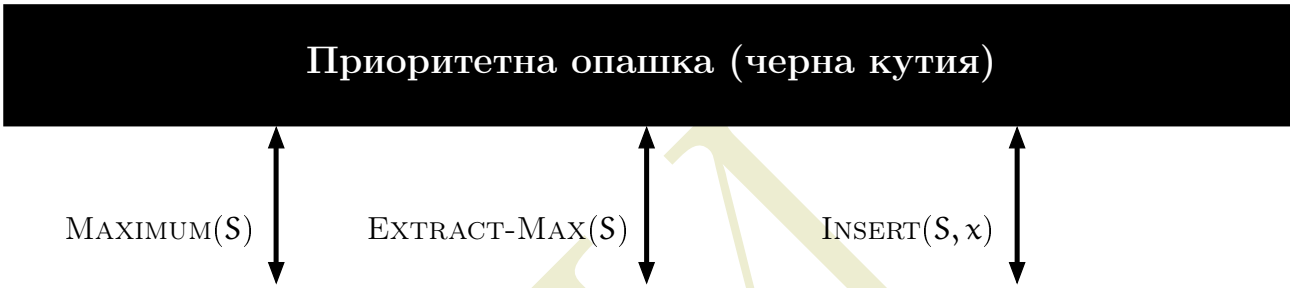
Приоритетна опашка (на английски, *priority queue*) е вид *абстрактен тип данни* (на английски, *abstract data type*, или ADT). Първо ще изясним понятието абстрактен тип данни (АТД). Според Sedgewick и Wayne [SW11, стр. 64]:

> *An* abstract data type *(ADT) is a data type whose representation is hidden from the client.*
> ...
> *When using an ADT, we focus on the operations specified in the API and pay no attention to the data representation; when implementing an ADT, we focus on the data, then implement operations on that data.*
>
> *To specify the behavior of an abstract data type, we use an* application programming interface *(API), ...*

И така, АТД е множество от данни, обикновено от един и същи тип, заедно с множество функции върху тях, което множество от функции се нарича *интерфейс*. "Абстрактен" означава, че конкретната реализация остава скрита, а цялото "общуване" с данните става през интерфейса. Конкретната реализация би могла да бъде променяна, като това остава прозрачно за софтуера, който ползва АТД-то (или поне на теория софтуерът-потребител не би трябвало да "усеща" разликата, ако няма бъгове и интерфейсът на новата реализация е изграден съгласно спецификациите). Конкретната реализация остава напълно скрита зад интерфейса и можем да си представяме АТД-то като черна кутия (*black box*) плюс интерфейс. Прости примери за АТД са *стек* и *опашка*, наричани още съответно LIFO и FIFO [SW11, стр. 121] (вж. Фигура 16 за илюстрация на черна кутия с интерфейс).

4.2 Приоритетна опашка: вид АТД

Приоритетните опашки имат елементи, всеки от които има две стойности: ключ и данни, свързани с този ключ. Ключовете обикновено са цели числа, макар че всеки тип данни, който може да се използва в контекста на сортирането, може да е типът данни на ключовете. Приоритетните опашки, подобно на пирамидите, са максимални и минимални. Тук ще дискутираме само максимални приоритетни опашки, като изложението за минималния тип е напълно аналогично. Казвайки "приоритетна опашка", имаме предвид максимална приоритетна опашка.



Фигура 16: АТД приоритетна опашка като черна кутия. Конкретната имплементация е скрита. Отвън се "виждат" само трите функции на интерфейса.

Интерфейсът на приоритетните опашки обикновено се ограничава до следните три функции. Ако приоритетната опашка се казва S, функциите от интерфейса са:

- Maximum(S): връща елемента с максимален ключ;
- Extract-Max(S): връща елемента с максимален ключ и го премахва от S;
- Insert(S, x): вмъква в S елемент x, който е от подходящ за S тип.

Написаното тук е според учебника на Cormen и др. [CLRS09, стр. 162]. Авторите дават и четвърта функция от интерфейса, а именно Increase-Key(S, x, k), която има смисъл на "увеличи ключа на x на нова стойност k, при допускането, че тя е не по-малка от стария ключ на x". Тази функция, която ще дискутираме след малко, не е част от стандартния интерфейс на приоритетните опашки. Sedgewick и Wayne [SW11, стр. 309] дават следния изчерпателен списък на интерфейса на реални (които се ползват на практика) приоритетни опашки. Описанието е на езиковите конвенции на Java.

- `MaxPQ()`: create a priority queue
- `MaxPQ(int max)`: create a priority queue of initial capacity `max`
- `MaxPQ(Key[] a)`: create a priority queue from the keys in `a[]`
- `void insert(Key v)`: insert a key into the priority queue
- `Key max()`: return the largest key
- `Key delMax()`: return and remove the largest key
- `boolean isEmpty()`: is the priority queue empty?
- `int size()`: number of keys in the priority queue

За целите на тази лекция, интерфейсът е според учебника на Cormen и др., като Increase-Key не е част от него, освен ако това не е казано изрично. Фигура 16 илюстрира приоритетна опашка като черна кутия плюс интерфейс.

Приложенията на приоритетните опашки са много. Всяка дейност, в която трябва да бъдат обработвани някакви неща, пристигащи асинхронно, като в даден момент може да се обработва само едно нещо, може да се моделира с приоритетна опашка. Нещата може да са, print jobs в операционна система, кораби за разтоварване в пристанище, самолети, кацащи на летище, спешно отделение в болница, и така нататък. С приоритетни опашки може и да се сортира: Heapsort можем да разглеждаме като сортиращ алгоритъм с последователно избиране на максимален елемент (какъвто е Selection Sort), ползващ ефикасно реализирана приоритетна опашка.

4.3 Реализация на приоритетни опашки с двоични пирамиди

Както ще видим след малко, двоичните пирамиди са много подходящи за реализация на приоритетни опашки, но в никакъв случай не са единствената възможна реализация. Приоритетна опашка може да бъде реализирана и чрез следните структури данни.

- Сортиран масив. На читателя остава да измисли имплементация на трите функции от интерфейса. Тук само ще спомена, че Maximum(S) и Extract-Max(S) при тази реализация имат сложност $\Theta(1)$, докато Insert(S, x) има сложност $\Theta(n)$.
- Несортиран масив. И тук остава на читателя да измисли имплементация на трите функции от интерфейса. При тази реализация, Maximum(S) и Extract-Max(S) имат сложност $\Theta(n)$, докато Insert(S, x) има сложност $\Theta(1)$.
- Пирамида на Фибоначи. Това е сложна структура данни, даваща възможност за изключително ефикасна реализация на функциите от интерфейса. Нещо повече, тази структура данни позволява интерфейс от по-голямо множество функции, които включват изтриване на елемент и смесване (merge) на две пирамиди. За съжаление, реализацията на функциите от интерфейса е ефикасна само в асимптотичния смисъл. Големите скрити константи правят пирамидите на Фибоначи неефикасни на практика, освен за наистина много големи данни. За подробности, вижте [CLRS09, стр. 506].
- Двоични дървета, реализирани чрез списъци с дървовидна структура (по два указателя към децата за всеки запис). Подробна дискусия на предимствата, недостатъците и вариациите на този подход има в третия том на култовата поредица на Доналд Кнут "Изкуството на Компютърното Програмиране" (The Art of Computer Programming) [Knu98, стр. 149]. От общи съображения е ясно, че този подход изисква значително—по-точно, линейно—повече памет от реализацията с масив. Измежду неговите предимства се откроява възможността за бързо сливане (merge) на приоритетни опашки.

Трите функции на интерфейса може да бъдат реализирани чрез двоични пирамиди по следния начин. Входът S е масив $S[1, \ldots, n]$, реализиращ пирамида. $S.size$ е размерът на пирамидата, тоест, само $S[1, \ldots, S.size]$ е пирамида. Допускаме, че винаги $S.size < n$, така че няма да се притесняваме за препълване на масива в Heap-Insert. Ако пирамидата е празна, както е в самото начало при създаването ѝ, $S.size = 0$.

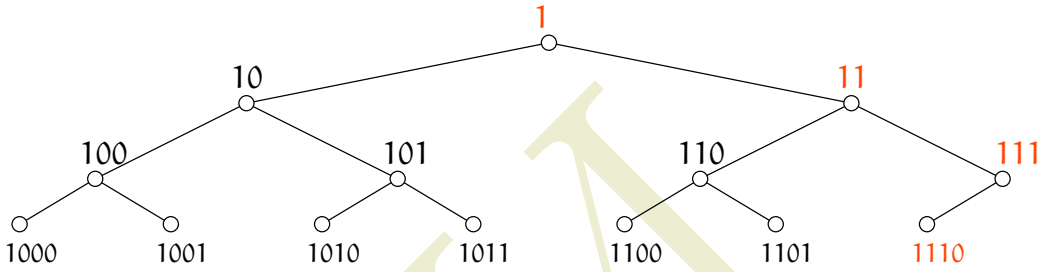
```
Heap-Maximum(S[1, ..., n])
1  (* Допускаме, че S.size ⩾ 1 *)
2  return S[1]
```

```
Heap-Extract-Max(S[1, ..., n])
1  (* Допускаме, че S.size ⩾ 1 *)
2  max ← S[1]
3  S[1] ← S[S.size]
4  S.size ← S.size − 1
5  Heapify(S, 1)
6  return max
```

```
Heap-Insert(S[1, ..., n], x)
1  S.size ← S.size + 1
2  i ← S.size
3  S[i] ← x
4  while i > 1 and S[Parent(i)] < S[i] do
5      swap(S[Parent(i)], S[i])
6      i ← Parent(i)
```

Коректността на тези функции е абсолютно очевидна след изложението в Секция 2. Сложността на Heap-Maximum е $\Theta(1)$, а на Heap-Extract-Max и Heap-Insert е $\Theta(\lg(S.size))$. Ако не ползваме "n" за големината на масива, а за големината на самата пирамида, и изразяваме сложността чрез символа "n", то двете функции имат сложност $\Theta(\lg n)$. Имайки предвид огромната, качествена разлика между нарастванията $\Theta(n)$ и $\Theta(\lg n)$, виждаме, че реализацията с двоични пирамиди е **много по-добра** от реализацията със сортирани или несортирани масиви.

Оттук правим важен извод, който е приложим в много по-широк контекст от ефикасното реализиране на приоритетните опашки. В някои случаи (реализиране на приоритетна опашка), оптималният подход (пирамида)



Фигура 17: Попълнено дърво с 14 върха и техните адреси. Специалните върхове са в червено.

е **компромис** (на английски, *tradeoff*) между два екстремални подхода (сортиран масив и несортиран масив). Относно две различни изисквания (бързо изваждане на максимален елемент и бързо вмъкване на нов елемен), при единият екстремален подход имаме оптимален резултат ($\Theta(1)$) по първото изискване, но изключително лош резултат ($\Theta(n)$) по второто; а при другия подход, оптимален резултат по второто изискване, но изключително лош резултат по първото. Компромисният подход (отказ от константно време за коя да е от операциите и постигане на задоволително време и за двете операции), направен интелигентно, се оказва печеливш.

4.4 Функцията Increase-Key

Според [CLRS09], функцията от интерфейса Increase-Key може да се реализира с двоична пирамида по следния начин.

```
Heap-Increase-Key(S[1,...,n], i, k)
1  if k < S[i]
2      error "опит да бъде намален ключът"
3  S[i] ← k
4  while i > 1 and S[Parent(i)] < S[i] do
5      swap(S[Parent(i)], S[i])
6      i ← Parent(i)
```

Очевидно, тази имплементация не може да е част от интерфейса на АТД, защото в нея е заложено, че имплементацията е именно чрез масив. В случая параметърът i е индексът на елемента, чийто ключ ще нараства. Ако имплементацията на пирамидата е чрез някакви свързани списъци, този индекс няма смисъл.

Правилното викане на функцията би било Increase-Key(S, x, k), където x е елемент от опашката. Но ако реализираме функцията с пирамида и дадем параметър x, който е елемент (с други думи, ако x е старата стойност на ключа, която трябва да стане k), този елемент трябва да бъде първо открит, което е операция с линейна сложност в пирамидите[†], дори ако допуснем уникални ключове. Така че, за да постигнем логаритмична сложност и на тази функция от интерфейса, се налага да "счупим" изискването за абстрактен тип данни и да заложим в нейната спецификация, че приоритетната опашка е реализирана именно чрез масив и че някак знаем индекса на елемента, чийто ключ искаме да повишим.

5 За броя на пирамидите

В тази секция ще разгледаме интересния въпрос, колко от всички $n!$ пермутации на n различни числа са пирамиди, по-точно, линеаризации на пирамиди. Точна формула за този брой има в [Knu98, стр. 152]. Да разгледаме отново попълненото двоично дърво с адресите от Подсекция 1.2, което за удобство показваме отново на Фигура 17.

Дървото има 14 върха. Да запишем броевете на върховете на поддърветата, вкоренени във всички върхове на дървото, в нарастващ ред по адресите на тези върхове. Ще записваме броевете в двоична позиционна бройна система:

$$1110, 111, 110, 11, 11, 11, 10, 1, 1, 1, 1, 1, 1, 1$$

[†]Примерно, най-малкият ключ може да е във всяко листо

За да е напълно ясно: цялото дърво (тоест, поддървото с корен върха, чийто адрес е 1) има четиринадесет върха, което в двоична позиционна бройна система е 1110. Поддървото, вкоренено в следващия връх (с адрес 10) има седем върха, така че записваме 111. И така нататък. С червен цвят са записани броевете на върховете в тези поддървета, чиито корени лежат на пътя между корена на дървото и последния връх (чийто адрес е $\mathsf{bin}(n)$). Тези върхове са наречени в [Knu98], *специалните върхове*, а дърветата, вкоренени в тях, *специалните поддървета*. В [Knu98, стр. 157, упр. 20 и 21] е показано, че ако $\mathsf{bin}(n) = b_k b_{k-1} \ldots b_1 b_0$, където $k = \lfloor \lg n \rfloor$ и $b_k = 1$, то размерите[†] на специалните поддървета, записани в двоична позиционна бройна система, са

$$1b_{k-1}\ldots b_1 b_0, \qquad 1b_{k-2}\ldots b_1 b_0, \qquad \ldots \qquad 1b_1 b_0, \qquad 1b_0, \qquad 1$$

а размерите на неспециалните поддървета числа от вида $2^m - 1$, защото те са съвършени двоични дървета, и броевете на неспециалните поддървета от всяка срещаща се големина са съответно

$$\begin{aligned}
&\left\lfloor \frac{n-1}{2} \right\rfloor \text{ числа } 1,\\
&\left\lfloor \frac{n-2}{4} \right\rfloor \text{ числа } 3,\\
&\left\lfloor \frac{n-4}{8} \right\rfloor \text{ числа } 7,\\
&\qquad \ldots\\
&\left\lfloor \frac{n-2^{k-1}}{2^k} \right\rfloor \text{ числа } 2^k - 1
\end{aligned}$$

Знаейки размерите на поддърветата, можем да изчислим H_n: броя на начините да разположим ключовете $\{1, 2, \ldots, n\}$ в пирамида, като използваме резултата от [Knu98, стр. 67, упр. 20]:

$$H_n = \frac{n!}{s_1 \times s_2 \times \ldots \times s_n} \tag{2}$$

където $\{s_1, s_2, \ldots, s_n\}_M$ е мултимножеството от всички различни размери на поддърветата. Примерно, ако елементите са 14, каквото е дървото на Фигура 17, начините да сложим ключовете $\{1, 2, \ldots, 14\}$, така че дървото да е пирамида, са на брой

$$H_{14} = \frac{14!}{14 \times 6 \times 2 \times 1 \times 1^6 \times 3^3 \times 7^1} = 2\,745\,600$$

Сравнете H_{14} с $14! = 87\,178\,291\,200$.

Редицата H_n за $n \geqslant 1$ е редица A056971 в онлайн енциклопедията на целочислените редици. Точният израз (2) не дава представа за асимптотиката на H_n. Асимптотиката на H_n се разглежда в [HMS01]. Основният резултат е твърде сложен, но грубо приближение е следното. Ако $h_n = \log\left(\frac{n!}{H_n}\right)$, то $h_n \asymp n$. Оттук—това е грубо приближение—асимптотиката на H_n е приблизително $\frac{n!}{c^n}$ за някаква константна c.

Литература

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Flo64] Robert W. Floyd. Algorithm 245: Treesort. *Communications of the ACM*, 7(12):701–, 1964.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[HMS01] Hsien-Kuei Hwang and Jean Marc Steyaert. On the number of heaps and the cost of heap construction, 2001. Достъпна онлайн на http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.9150&rep=rep1&type=pdf.

[†] Това са общо $k = \lfloor \lg n \rfloor$ числа, колкото е височината на попълненото двоично дърво с n върха съгласно Лема 1.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[NIS] tree (data structure). National Institute of Standards and Technology's web site. URL http://xlinux.nist.gov/dads/HTML/tree.html .

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.