1. Datové struktury

V algoritmech potřebujeme zacházet s různými druhy dat – posloupnostmi, množinami, grafy, … Často se nabízí více způsobů, jak tato data uložit do paměti počítače. Jednotlivé způsoby se mohou lišit spotřebou paměti, ale také rychlostí různých operací s daty. Vhodný způsob tedy volíme podle toho, jaké operace využívá konkrétní algoritmus a jak často je provádí.

Otázky tohoto druhu se přitom opakují. Proto je zkoumáme obecně, což vede ke studiu datových struktur. V této kapitole se podíváme na ty nejběžnější z nich.

1.1. Rozhraní datových struktur

Nejprve si rozmyslíme, co od datové struktury očekáváme. Z pohledu programu má struktura jasné rozhraní: reprezentuje nějaký druh dat a umí s ním provádět určité operace. Uvnitř datové struktury pak volíme konkrétní uložení dat v paměti a algoritmy pro provádění jednotlivých operací. Z toho pak plyne prostorová složitost struktury a časová složitost operací.

Fronta a zásobník

Jednoduchým příkladem je *fronta.* Ta si pamatuje posloupnost prvků a umí s ní provádět tyto operace:

ENQUEUE(x)	přidá na konec fronty prvek x
DEQUEUE	odebere prvek ze začátku fronty, případně oznámí, že fronta je prázdná

Pokud frontu implementujeme jako spojový seznam, zvládneme obě operace v konstantním čase a vystačíme si s pamětí lineární v počtu prvků.

Nejbližším příbuzným fronty je *zásobník* – ten si také pamatuje posloupnost a dovede přidávat nové prvky na konec, ale odebírá je z téhož konce. Operaci přidání se obvykle říká PUSH, operaci odebrání POP.

Prioritní fronta

Zajímavější je „fronta s předbíháním“, zvaná též *prioritní fronta.* Každý prvek má přiřazenu číselnou *prioritu* a na řadu vždy přijde prvek s nejvyšší prioritou. Operace vypadají následovně:

ENQUEUE(x, p)	přidá do fronty prvek x s prioritou p
DEQUEUE	nalezne prvek s nejvyšší prioritou a odebere ho (pokud je takových prvků víc, vybere libovolný z nich)

Prioritní frontu lze reprezentovat polem nebo seznamem, ale nalezení maxima z priorit bude pomalé – v n-prvkové frontě $\Theta(n)$. V oddílu 1.2 zavedeme *haldu,* s níž dosáhneme časové složitosti $\mathcal{O}(\log n)$ u obou operací.

Množina a slovník

Množina obsahuje konečný počet prvků vybraných z nějakého *univerza*. Pod univerzem si můžete představit třeba celá čísla, ale nemusíme se omezovat jen na ně. Obecně budeme předpokládat, že prvky univerza je možné v konstantním čase přiřazovat a porovnávat na rovnost a „je menší než“.

Množina nabízí následující operace:

MEMBER(x)	zjistí, zda x leží v množině (někdy též FIND(x))
INSERT(x)	vloží x do množiny (pokud tam už bylo, nestane se nic)
DELETE(x)	odebere x z množiny (pokud tam nebylo, nestane se nic)

Zobecněním množiny je *slovník.* Ten si pamatuje konečnou množinu *klíčů* a každému z nich přiřazuje *hodnotu* (to může být prvek nějakého dalšího univerza, nebo třeba ukazatel na jinou datovou strukturu). Slovník je tedy konečná množina dvojic (*klíč*, *hodnota*), v níž se neopakují klíče. Typické slovníkové operace jsou tyto:

GET(x)	zjistí, jaká hodnota je přiřazena klíči x (pokud nějaká)
SET(x, y)	přiřadí klíči x hodnotu y; pokud už nějaká dvojice s klíčem x existovala, tak ji nahradí
DELETE(x)	smaže dvojici s klíčem x (pokud existovala)

Někdy nás také zajímá vzájemné pořadí prvků – tehdy definujeme *uspořádanou množinu,* která má navíc tyto operace:

MIN	vrátí nejmenší hodnotu v množině
MAX	vrátí největší hodnotu v množině
PRED(x)	vrátí největší prvek menší než x, nebo řekne, že takový není
SUCC(x)	vrátí nejmenší prvek větší než x, nebo řekne, že takový není

Obdobně můžeme zavést uspořádané slovníky.

Množinu nebo slovník můžeme reprezentovat pomocí pole. Má to ale své nevýhody: Především potřebujeme dopředu znát horní mez počtu prvků množiny, případně si pořídit „nafukovací“ pole (viz oddíl **??**). Mimo to se s polem pracuje pomalu: množinové operace musí pokaždé projít všechny prvky, což trvá $\Theta(n)$.

Hledání můžeme zrychlit *uspořádáním* (setříděním) pole. Pak může MEMBER binárně vyhledávat v logaritmickém čase, ovšem vkládání i mazání zůstanou lineární.

Použijeme-li *spojový seznam,* všechny operace budou lineární. Uspořádáním seznamu si nepomůžeme, protože v seznamu nelze hledat binárně.

Můžeme trochu podvádět a upravit rozhraní. Kdybychom slíbili, že INSERT nikdy nezavoláme na prvek, který už v množině leží, mohli bychom nový prvek vždy přidat na konec pole či seznamu. Podobně kdyby DELETE dostal místo klíče ukazatel na už nalezený prvek, mohli bychom mazat v konstantním čase (cvičení 2).

Později vybudujeme vyhledávací stromy (kapitola **??**) a hešovací tabulky (oddíl **??**), které budou mnohem efektivnější. Abyste věděli, na co se těšit, prozradíme už teď složitosti jednotlivých operací:

	INSERT	DELETE	MEMBER	MIN	PRED
pole	$\Theta(1)^*$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
uspořádané pole	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$
spojový seznam	$\Theta(1)^*$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
uspořádaný seznam	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
vyhledávací strom	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
hešovací tabulka	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	$\Theta(n)$	$\Theta(n)$

Operace MAX a SUCC jsou stejně rychlé jako MIN a PRED. Složitosti označené hvězdičkou platí jen tehdy, slíbíme-li, že se prvek v množině dosud nenachází; v opačném případě je potřeba předem provést MEMBER. Složitosti označené křížkem jsou průměrné hodnoty. U polí předpokládáme, že dopředu známe horní odhad velikosti množiny.

Cvičení

1. Navrhněte reprezentaci fronty v poli, která bude pracovat v konstantním čase. Můžete předpokládat, že předem znáte horní odhad počtu prvků ve frontě.
2. Při mazání z pole vznikne díra, kterou je potřeba zaplnit. Jak to udělat v konstantním čase?
3. Množiny čísel můžeme reprezentovat uspořádanými seznamy. Ukažte, jak v této reprezentaci počítat průnik a sjednocení množin v lineárním čase.

4.* Na uspořádané množině můžeme také definovat operaci INDEX(i), která najde i-tý nejmenší prvek. K ní inverzní je operace RANK(x), jež spočítá počet prvků množiny menších než x. Rozmyslete, jakou složitost tyto dvě operace mají v různých reprezentacích množin.

1.2. Haldy

Jednou z nejjednodušších datových struktur je *halda* (anglicky *heap*), přesněji řečeno *minimová binární halda.* Co taková halda umí? Pamatuje si množinu prvků opatřených klíči a v nejjednodušší variantě nabízí tyto operace:

INSERT(x)	vloží prvek x do množiny
FINDMIN(x)	najde prvek s nejmenším klíčem (pokud je takových víc, pak libovolný z nich)
EXTRACTMIN(x)	odebere prvek s nejmenším klíčem a vrátí ho jako výsledek

Klíč přiřazený prvku x budeme značit $k(x)$. Klíče si můžeme představovat jako celá čísla, ale obecně to mohou být prvky libovolného univerza. Jako obvykle budeme předpokládat, že klíče lze přiřazovat a porovnávat v konstantním čase.

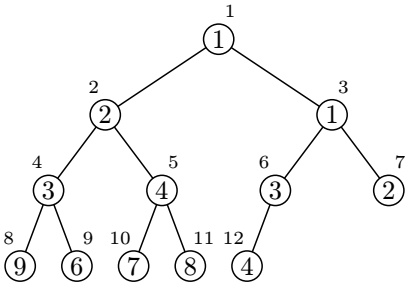
Definice: Strom nazveme *binární*, pokud je zakořeněný a každý vrchol má nejvýše dva syny, u nichž rozlišujeme, který je levý a který pravý. Vrcholy rozdělíme podle vzdálenosti od kořene do *hladin:* v nulté hladině leží kořen, v první jeho synové atd.

Definice: *Minimová binární halda* je datová struktura tvaru binárního stromu, v jehož každém vrcholu je uložen jeden prvek. Přitom platí:

1. *Tvar haldy:* Všechny hladiny kromě poslední jsou plně obsazené. Poslední hladina je zaplněna zleva.
2. *Haldové uspořádání:* Je-li v vrchol a s jeho syn, platí $k(v) \leq k(s)$.

Pozorování: Vydáme-li se z kořene dolů po libovolné cestě, klíče nemohou klesat. Proto se v kořeni stromu musí nacházet jeden z minimálních prvků (těch s nejmenším klíčem; kdykoliv budeme mluvit o porovnávání prvků, myslíme tím podle klíčů).

Podotýkáme ještě, že haldové uspořádání popisuje pouze „svislé“ vztahy. Například o relaci mezi levým a pravým synem téhož vrcholu pranic neříká.



Obr. 1.1: Halda a její očíslování

Lemma: Halda s n prvky má $\lfloor \log_2 n \rfloor + 1$ hladin.

Důkaz: Nejprve spočítáme, kolik vrcholů obsahuje binární strom o h úplně plných hladinách: $2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = 2^h - 1$. Pokud tedy do haldy přidáváme vrcholy po hladinách, nová hladina přibude pokaždé, když počet vrcholů dosáhne mocniny dvojky. □

Haldu jsme sice definovali jako strom, ale díky svému pravidelnému tvaru může být v paměti počítače uložena mnohem jednodušším způsobem. Vrcholy stromu očíslujeme *indexy* $1, \ldots, n$. Číslovat budeme po hladinách shora dolů, každou hladinu zleva doprava. V tomto pořadí můžeme vrcholy uložit do pole a pracovat s nimi jako se stromem. Platí totiž:

Pozorování: Má-li vrchol index i, pak jeho levý syn má index $2i$ a pravý syn $2i + 1$. Je-li $i > 1$, otec vrcholu i má index $\lfloor i/2 \rfloor$, přičemž $i \bmod 2$ nám řekne, zda je k otci připojen levou, nebo pravou hranou.

Zatím budeme předpokládat, že dopředu víme, kolik prvků budeme do haldy vkládat, a podle toho zvolíme velikost pole. Později (v oddílu **??**) ukážeme, jak lze haldu podle potřeby zmenšovat a zvětšovat.

Dodejme ještě, že obdobně můžeme zavést *maximovou haldu,* která používá opačné uspořádání, takže namísto minima umí rychle najít maximum. Všechno, co v této kapitole ukážeme pro minimovou haldu, platí analogicky i pro tu maximovou.

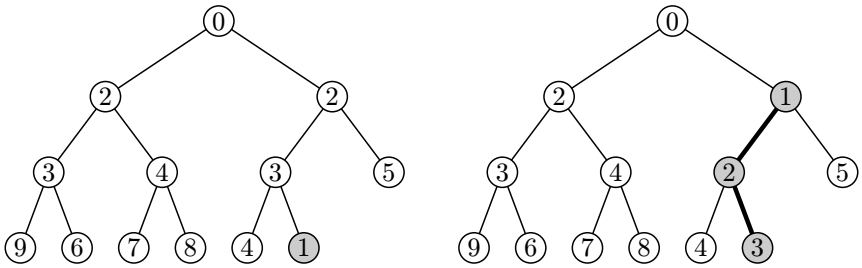
Vkládání

Prázdnou nebo jednoprvkovou haldu vytvoříme triviálně. Uvažujme nyní, jak do haldy přidat další prvek.

Podmínka na tvar haldy nám dovoluje přidat nový list na konec poslední hladiny. Pokud by tato hladina byla plná, můžeme založit novou s jediným listem úplně vlevo. Tím dostaneme strom správného tvaru, ale na hraně mezi novým listem ℓ a jeho otcem o jsme mohli porušit uspořádání, pokud $k(\ell) < k(o)$.

V takovém případě list s otcem prohodíme. Tím jsme chybu opravili, ale mohli jsme způsobit další chybu o hladinu výš. Tu vyřešíme dalším prohozením a tak dále. Nově přidaný prvek bude tedy „vybublávat“ nahoru, potenciálně až do kořene.

Zbývá se přesvědčit, že kdykoliv jsme prohodili otce se synem, nemohli jsme pokazit vztah mezi otcem a jeho druhým synem. To proto, že otec se prohozením zmenšil.



Obr. 1.2: Vkládání do haldy: začátek a konec

Nyní vkládání zapíšeme v pseudokódu. Budeme předpokládat, že halda je uložena v poli, takže na všechny vrcholy se budeme odkazovat indexy. Prvek na indexu i označíme $p(i)$ a jeho klíč $k(i)$, v proměnné n si budeme pamatovat momentální velikost haldy.

Procedura HeapInsert (vkládání do haldy)
Vstup: Nový prvek p s klíčem k

1. $n \leftarrow n + 1$
2. $p(n) \leftarrow p$, $k(n) \leftarrow k$
3. BubbleUp(n)

Procedura BubbleUp(i)
Vstup: Index i vrcholu se změněným klíčem

1. Dokud $i > 1$:

2. $o \leftarrow \lfloor i/2 \rfloor$ ◁ *otec vrcholu* i
3. Je-li $k(o) \leq k(i)$, vyskočíme z cyklu.
4. Prohodíme $p(i)$ s $p(o)$ a $k(i)$ s $k(o)$.
5. $i \leftarrow o$

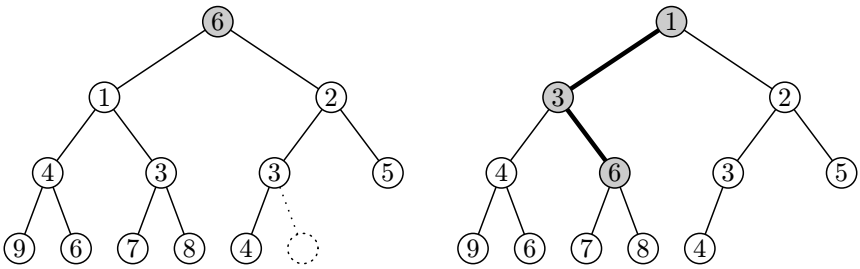
Časovou složitost odhadneme snadno: na každé hladině stromu strávíme nejvýše konstantní čas a hladin je logaritmický počet. Operace INSERT tedy trvá $\mathcal{O}(\log n)$.

Hledání a mazání minima

Operace nalezení minima (FINDMIN) je triviální, stačí se podívat do kořene haldy, tedy na index 1. Zajímavější bude, když se nám zachce provést EXTRACTMIN čili minimum odebrat.

Kořen stromu přímo odstranit nemůžeme. Axiom o tvaru haldy nám nicméně dovoluje beztrestně smazat nejpravější vrchol na nejnižší hladině. Smažeme tedy ten a prvek, který tam byl uložený, přesuneme do kořene.

Opět jsme v situaci, kdy strom má správný tvar, ale může mít pokažené uspořádání. Konkrétně se mohlo stát, že nový prvek v kořeni je větší než některý z jeho synů, možná dokonce než oba. V takovém případě kořen prohodíme s menším z obou synů. Tím jsme opravili vztahy mezi kořenem a jeho syny, ale mohli jsme způsobit obdobný problém o hladinu níže. Pokračujeme tedy stejným způsobem a „zabubláváme“ nově vložený prvek hlouběji, potenciálně až do listu.



Obr. 1.3: Mazání z haldy: začátek a konec

Procedura HEAPEXTRACTMIN (mazání minima z haldy)

1. $p \leftarrow p(1)$, $k \leftarrow k(1)$
2. $p(1) \leftarrow p(n)$, $k(1) \leftarrow k(n)$
3. $n \leftarrow n - 1$
4. BUBBLEDOWN(1)

Výstup: Prvek p s minimálním klíčem k

Procedura BUBBLEDOWN(i)

Vstup: Index i vrcholu se změněným klíčem

1. Dokud $2i \leq n$: ◁ *vrchol* i *má nějaké syny*

2. $s \leftarrow 2i$
3. Pokud $s+1 \leq n$ a $k(s+1) < k(s)$:
4. $\quad s \leftarrow s+1$
5. Pokud $k(i) < k(s)$, vyskočíme z cyklu.
6. Prohodíme $p(i)$ s $p(s)$ a $k(i)$ s $k(s)$.
7. $i \leftarrow s$

Opět trávíme čas $\mathcal{O}(1)$ na každé hladině, celkem tedy $\mathcal{O}(\log n)$.

Úprava klíče

Doplňme ještě jednu operaci, která se nám bude časem hodit. Budeme jí říkat Decrease a bude mít za úkol snížit klíč prvku, jenž v haldě už je.

Tvar kvůli tomu měnit nemusíme, ale co se stane s uspořádáním? Směrem dolů jsme ho pokazit nemohli, směrem nahoru ano. Jsme tedy ve stejné situaci jako při InsertUp, takže stačí zavolat proceduru BubbleUp, aby uspořádání opravila. To stihneme v logaritmickém čase.

Je tu ale jeden zádrhel: musíme vědět, kde se prvek v haldě nachází. Podle klíče vyhledávat neumíme, ovšem můžeme haldu naučit, aby nás informovala, kdykoliv se změní poloha nějakého prvku.

Obdobně můžeme implementovat zvýšení klíče (Increase). Uspořádání se tentokrát bude kazit směrem dolů, takže ho budeme opravovat bubláním v tomto směru.

Všimněte si, že Insert můžeme ekvivalentně popsat jako přidání listu s hodnotou $+\infty$ a následný Decrease. Podobně ExtractMin odpovídá smazání listu a Increase kořene.

Složitost haldových operací shrneme následující větou:

Věta: V binární haldě o n prvcích trvají operace Insert, ExtractMin, Increase a Decrease čas $\mathcal{O}(\log n)$. Operace FindMin trvá $\mathcal{O}(1)$.

Konstrukce haldy

Pomocí haldy můžeme třídit: vytvoříme prázdnou haldu, do ní InsertUjeme tříděné prvky a pak je pomocí ExtractMin vytahujeme od nejmenšího po největší. Jelikož provedeme $2n$ operací s nejvýše n-prvkovou haldou, má tento třídicí algoritmus časovou složitost $\mathcal{O}(n \log n)$.

Samotné vytvoření n-prvkové haldy lze dokonce stihnout v čase $\mathcal{O}(n)$. Provedeme to následovně: Nejprve prvky rozmístíme do vrcholů binárního stromu v libovolném pořadí – pokud máme strom uložený v poli, nemuseli jsme pro to udělat vůbec nic, prostě jenom začneme pozice v poli chápat jako indexy ve stromu.

Pak budeme opravovat uspořádání od nejnižší hladiny až k té nejvyšší, tedy v pořadí klesajících indexů. Kdykoliv zpracováváme nějaký vrchol, využijeme toho, že celý podstrom pod ním je už uspořádaný korektně, takže na opravu vztahů mezi novým vrcholem a jeho syny stačí provést bublání dolů. Pseudokód je mile jednoduchý:

Procedura MakeHeap (konstrukce haldy)
Vstup: Posloupnost prvků $x_1, \ldots, x_n$ s klíči $k_1, \ldots, k_n$

1. Prvky uložíme do pole tak, že $x(i) = x_i$ a $k(i) = k_i$.
2. Pro $i = \lfloor n/2 \rfloor, \ldots, 1$:
3. BubbleDown(i)

Výstup: Halda

Věta: Operace MakeHeap má časovou složitost $\mathcal{O}(n)$.

Důkaz: Nechť strom má h hladin očíslovaných od 0 (kořen) do $h-1$ (listy). Bez újmy na obecnosti budeme předpokládat, že všechny hladiny jsou úplně plné, takže $n = 2^h - 1$.

Zprvu se zdá, že provádíme n bublání, která trvají logaritmicky dlouho, takže jimi strávíme čas $\Theta(n \log n)$. Podíváme-li se pozorněji, všimneme si, že například na hladině $h-2$ leží přibližně $n/4$ prvků, ale každý z nich bubláme nejvýše o hladinu níže. Intuitivně většina vrcholů leží ve spodní části stromu, kde s nimi máme málo práce. Nyní to řekneme exaktněji.

Jedno BubbleDown na i-té hladině trvá $\mathcal{O}(h-1-i)$. Pokud to sečteme přes všech 2^i vrcholů hladiny a poté přes všechny hladiny, dostaneme (až na konstantu z $\mathcal{O}$):

$$\sum_{i=0}^{h-1} 2^i \cdot (h-1-i) = \sum_{j=0}^{h-1} 2^{h-1-j} \cdot j = \sum_{j=0}^{h-1} \frac{2^{h-1}}{2^j} \cdot j \leq n \cdot \sum_{j=0}^{h-1} \frac{j}{2^j} \leq n \cdot \sum_{j=0}^{\infty} \frac{j}{2^j}.$$

Podle podílového kritéria konvergence řad poslední suma konverguje, takže předposlední suma je shora omezena konstantou. □

Poznámka: Argument s konvergencí řady zaručuje existenci konstanty, ale její hodnota by mohla být absurdně vysoká. Pochybnosti zaplašíme sečtením řady. Jde to provést hezkým trikem – místo nekonečné řady budeme sčítat nekonečnou matici:

$$\begin{pmatrix} 1/2 & & & \\ 1/4 & 1/4 & & \\ 1/8 & 1/8 & 1/8 & \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Sčítáme-li její prvky po řádcích, vyjde hledaná suma $\sum_j j/2^j$. Nyní budeme sčítat po sloupcích: první sloupec tvoří geometrickou řadu s kvocientem $1/2$, a tedy součtem 1 (to je mimochodem hezky vidět z binárního zápisu: $0.1 + 0.01 + 0.001 + \ldots = 0.111\ldots = 1$). Druhý sloupec má poloviční součet, třetí čtvrtinový, atd. Součet součtů sloupců je tudíž $1 + 1/2 + 1/4 + \ldots = 2$.

Třídění haldou – Heapsort

Již jsme přišli na to, že pomocí haldy lze třídit. Potřebovali jsme na to ovšem lineární pomocnou paměť na uložení haldy. Nyní ukážeme elegantnější a úspornější algoritmus, kterému se obvykle říká Heapsort.

Vstup dostaneme v poli. Z tohoto pole vytvoříme operací MakeHeap *maximovou* haldu. Pak budeme opakovaně mazat maximum. Halda se bude postupně zmenšovat a uvolněné místo v poli budeme zaplňovat setříděnými prvky.

Obecně po k-tém kroku bude na indexech $1, \ldots, n-k$ uložena halda a na $n-k+1, \ldots, n$ bude ležet posledních k prvků setříděné posloupnosti. V dalším kroku se tedy maximum haldy přesune na pozici $n-k$ a hranice mezi haldou a setříděnou posloupností se posune o 1 doleva.

Algoritmus HeapSort (třídění haldou)
Vstup: Pole $x_1, \ldots, x_n$

1. Pro $i = \lfloor n/2 \rfloor, \ldots, 1$: ◁ *vytvoříme z pole maximovou haldu*
2. HsBubbleDown(n, i)
3. Pro $i = n, \ldots, 2$:
4. Prohodíme x_1 s x_i. ◁ *maximum se dostane na správné místo*
5. HsBubbleDown$(i-1, 1)$ ◁ *opravíme haldu*

Výstup: Setříděné pole $x_1, \ldots, x_n$

Bublací procedura funguje podobně jako BubbleDown, jen používá opačné uspořádání a nerozlišuje prvky od jejich klíčů.

Procedura HsBubbleDown(m, i)
Vstup: Aktuální velikost haldy m, index vrcholu i

1. Dokud $2i \leq m$:
2. $s \leftarrow 2i$
3. Pokud $s+1 \leq m$ a $x_{s+1} > x_s$:
4. $s \leftarrow s+1$
5. Pokud $x_i > x_s$, vyskočíme z cyklu.
6. Prohodíme x_i a x_s.
7. $i \leftarrow s$

Věta: Algoritmus HeapSort setřídí n prvků v čase $\mathcal{O}(n \log n)$.

Důkaz: Celkem provedeme $\mathcal{O}(n)$ volání procedury HsBubbleDown. V každém okamžiku je v haldě nejvýše n prvků, takže jedno bublání trvá $\mathcal{O}(\log n)$. □

Z toho, že umíme pomocí haldy třídit, také plyne, že časová složitost haldových operací je nejlepší možná:

Věta: Mějme datovou strukturu s operacemi Insert a ExtractMin, která prvky pouze porovnává a přiřazuje. Pak má na n-prvkové množině alespoň jedna z těchto operací složitost $\Omega(\log n)$.

Důkaz: Pomocí n volání Insert a n volání ExtractMin lze setřídit n-prvkovou posloupnost. Z oddílu **??** ale víme, že každý třídicí algoritmus v porovnávacím modelu má složitost $\Omega(n \log n)$. □

Cvičení

1. Prioritní fronta se někdy definuje tak, že prvky se stejnou prioritou vrací v pořadí, v jakém byly do fronty vloženy. Ukažte, jak takovou frontu realizovat pomocí haldy. Dosáhněte časové složitosti $\mathcal{O}(\log n)$ na operaci.
2. Navrhněte operaci DELETE, která z haldy smaže prvek zadaný svým indexem.
3.* Dokažte, že vyhledávání prvku v haldě podle klíče vyžaduje čas $\Theta(n)$.
4. Definujme *d-regulární haldu* jako d-ární strom, který splňuje stejné axiomy o tvaru a uspořádání jako binární halda (binární tedy znamená totéž co 2-regulární). Ukažte, že d-ární strom má hloubku $\mathcal{O}(\log_d n)$ a lze ho také kódovat do pole. Dokažte, že haldové operace bublající nahoru trvají $\mathcal{O}(\log_d n)$ a ty bublající dolů $\mathcal{O}(d \log_d n)$. Zvýšením d tedy můžeme zrychlit INSERT a DECREASE za cenu zpomalení EXTRACTMIN a INCREASE. To se bude hodit v Dijkstrově algoritmu, viz cvičení **??**.
5. V rozboru operace MAKEHEAP jsme přehazovali pořadí sčítání v nekonečném součtu. To obecně nemusí být ekvivalentní úprava. Využijte poznatků z matematické analýzy, abyste dokázali, že v tomto případě se není čeho bát.

1.3. Písmenkové stromy

Další jednoduchou datovou strukturou je *písmenkový strom* neboli *trie.*[1] Slouží k uložení *slovníku* nejen podle naší definice z oddílu 1.1, ale i v běžném smyslu tohoto slova. Pamatuje si množinu *slov* – řetězců složených ze znaků nějaké pevné konečné abecedy – a každému slovu může přiřadit nějakou hodnotu (třeba překlad slova do kočkovštiny).

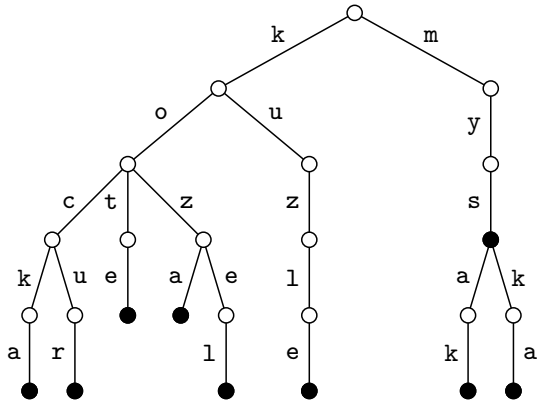
Trie má tvar zakořeněného stromu. Z každého vrcholu vedou hrany označené navzájem různými znaky abecedy. V kořeni odpovídají prvnímu písmenu slova, o patro níž druhému, a tak dále.

Vrcholům můžeme přiřadit řetězce tak, že přečteme všechny znaky na cestě z kořene do daného vrcholu. Kořen bude odpovídat prázdnému řetězci, čím hlouběji půjdeme, tím delší budou řetězce. Vrcholy odpovídající slovům slovníku označíme a uložíme do nich hodnoty přiřazené klíčům. Všimněte si, že označené mohou být i vnitřní vrcholy, je-li jeden klíč pokračováním jiného.

Každý vrchol si tedy bude pamatovat pole ukazatelů na syny (jako indexy slouží znaky abecedy), dále jednobitovou značku, zda se jedná o slovo ve slovníku, a případně hodnotu přiřazenou tomuto slovu. Je-li abeceda konstantně velká, celý vrchol zabere konstantní prostor. Pro větší abecedu můžeme pole nahradit některou z množinových datových struktur z příštích kapitol.

Vyhledávání (operace MEMBER) bude probíhat takto: Začneme v kořeni a budeme následovat hrany podle jednotlivých písmen hledaného slova. Pokud budou

[1] Zvláštní název, že? Vznikl zkřížením slov *tree* (strom) a *retrieval* (vyhledávání). Navzdory angličtině se u nás vyslovuje „trije“ a skloňuje podle vzoru růže.



Obr. 1.4: Písmenkový strom pro slova `kocka`, `kocur`, `kote`, `koza`, `kozel`, `kuzle`, `mys`, `mysak`, `myska`

všechny existovat, stačí se podívat, jestli vrchol, do kterého jsme došli, obsahuje značku. Chceme-li kdykoliv jít po neexistující hraně, ihned odpovíme, že se slovo se slovníku nenachází. Časová složitost hledání je lineární s délkou hledaného slova. Všimněte si, že na rozdíl od jiných datových struktur složitost nezávisí na tom, v jak velkém slovníku hledáme.

Při přidávání slova (operace INSERT) se nové slovo pokusíme vyhledat. Kdykoliv při tom bude nějaká hrana chybět, vytvoříme ji a necháme ji ukazovat na nový list. Vrchol, do kterého nakonec dojdeme, opatříme značkou. Časová složitost je zřejmě lineární s délkou přidávaného slova.

Při mazání (operace DELETE) bychom mohli slovo vyhledat a pouze z jeho koncového vrcholu odstranit značku. Tím by se nám ale mohly začít hromadit větve, které už nevedou do žádných označených vrcholů, a tedy jsou zbytečné. Proto mazání naprogramujeme rekurzivně: nejprve budeme procházet stromem dolů a hledat slovo, pak smažeme značku a budeme se vracet do kořene. Kdykoliv přitom narazíme na vrchol, který nemá ani značku, ani syny, smažeme ho. I zde vše stihneme v lineárním čase s délkou slova.

Vytvořili jsme tedy datovou strukturu pro reprezentaci slovníku řetězců, která zvládne operace MEMBER, INSERT a DELETE v lineárním čase s počtem znaků operandu. Jelikož stále platí, že všechny vrcholy stromu odpovídají prefixům (začátkům) slov ve slovníku, spotřebujeme prostor nejvýše lineární se součtem délek slovníkových slov.

Cvičení

1. Zkuste v písmenkovém stromu na obrázku vyhledat slova `kocka`, `kot` a `myval`. Pak přidejte `kot` a `kure` a nakonec smažte `myska`, `mysak` a `mys`.
2. Vymyslete, jak pomocí písmenkového stromu setřídit posloupnost řetězců v čase

lineárním vzhledem k součtu jejich délek. Porovnejte s algoritmem přihrádkového třídění z oddílu **??**.

3. Je dán text rozdělený na slova. Chceme vypsat frekvenční slovník, tedy tabulku všech slov setříděných podle počtu výskytů.
4. Vymyslete, jak pomocí písmenkového stromu reprezentovat množinu celých čísel z rozsahu 1 až ℓ. Jak bude složitost operací záviset na ℓ a na velikosti množiny?
5. Navrhněte datovou strukturu pro básníky, která si bude pamatovat slovník a bude umět hledat rýmy. Tedy pro libovolné zadané slovo najde jiné slovo ve slovníku, které má se zadaným co nejdelší společný suffix.

6.* Upravte datovou strukturu z předchozího cvičení, aby v případě, že nejlepších rýmů je více, vypsala lexikograficky nejmenší z nich.

7. Jak reprezentovat slovník, abyste uměli rychle vyhledávat všechny přesmyčky zadaného slova?
8. *Komprese trie:* Písmenkové stromy často obsahují dlouhé nevětvící se cesty. Tyto cesty můžeme komprimovat: nahradit jedinou hranou, která bude namísto jednoho písmene popsána celým řetězcem. Nadále bude platit, že všechny hrany vycházející z jednoho vrcholu se liší v prvních písmenech. Dokažte, že komprimovaná trie pro množinu n slov má nejvýše $\mathcal{O}(n)$ vrcholů. Upravte operace MEMBER, INSERT a DELETE, aby fungovaly v komprimované trii.

1.4. Prefixové součty

Nyní se budeme zabývat datovými strukturami pro *intervalové operace.* Obecně se tím myslí struktury, které si pamatují nějakou posloupnost prvků $x_1, \ldots, x_n$ a dovedou efektivně zacházet se souvislými podposloupnostmi typu $x_i, x_{i+1}, \ldots, x_j$. Těm se obvykle říká *úseky* nebo také *intervaly* (anglicky *range*).

Začneme elementárním příkladem: Dostaneme posloupnost a chceme umět odpovídat na dotazy typu „Jaký je součet daného úseku?“. K tomu se hodí spočítat takzvané prefixové součty:

Definice: *Prefixové součty* pro posloupnost $x_1, \ldots, x_n$ tvoří posloupnost $p_1, \ldots, p_n$, kde $p_i = x_1 + \ldots + x_i$. Obvykle se hodí položit $p_0 = 0$ jako součet prázdného prefixu.

Všechny prefixové součty si dovedeme pořídit v čase $\Theta(n)$, jelikož $p_0 = 0$ a $p_{i+1} = p_i + x_{i+1}$. Jakmile je máme, hravě spočítáme součet obecného úseku $x_i + \ldots + x_j$: můžeme ho totiž vyjádřit jako rozdíl dvou prefixových součtů $p_j - p_{i-1}$.

Naše datová struktura tedy spotřebuje čas $\Theta(n)$ na inicializaci a pak dokáže v čase $\Theta(1)$ odpovídat na dotazy. Prvky posloupnosti nicméně neumí měnit – snadno si rozmyslíme, že změna prvku x_1 způsobí změnu všech prefixových součtů. Takovým strukturám se říká *statické,* na rozdíl od *dynamických*, jako je třeba halda.

Rozklad na bloky

Existuje řada technik, jimiž lze ze statické datové struktury vyrobit dynamickou. Jednu snadnou metodu si nyní ukážeme. V zájmu zjednodušení notace posuneme indexy tak, aby posloupnost začínala prvkem x_0.

Vstup rozdělíme na bloky velikosti b (konkrétní hodnotu b zvolíme později). První blok bude tvořen prvky $x_0, \ldots, x_{b-1}$, druhý prvky $x_b, \ldots, x_{2b-1}$, atd. Celkem tedy vznikne n/b bloků. Pakliže n není dělitelné b, doplníme posloupnost nulami na celý počet bloků.

Obr. 1.5: Rozklad na bloky pro $n = 30$, $b = 6$ a dva dotazy

Libovolný zadaný úsek se buďto celý vejde do jednoho bloku, nebo ho můžeme rozdělit na konec (suffix) jednoho bloku, nějaký počet celých bloků a začátek (prefix) dalšího bloku. Libovolná z těchto částí přitom může být prázdná.

Pořídíme si tedy dva druhy struktur:

- *Lokální* struktury $L_1, \ldots, L_{n/b}$ budou vyřizovat dotazy uvnitř bloku. K tomu nám stačí spočítat pro každý blok prefixové součty.
- *Globální* struktura G bude naopak pracovat s celými bloky. Budou to prefixové součty pro posloupnost, která vznikne nahrazením každého bloku jediným číslem – jeho součtem.

Inicializaci těchto struktur zvládneme v lineárním čase: Každou z n/b lokálních struktur vytvoříme v čase $\Theta(b)$. Pak spočteme $\Theta(n/b)$ součtů bloků, každý v čase $\Theta(b)$ – nebo se na ně můžeme zeptat lokálních struktur. Nakonec vyrobíme globální strukturu v čase $\Theta(n/b)$. Všechno dohromady trvá $\Theta(n/b \cdot b) = \Theta(n)$.

Každý dotaz na součet úseku nyní můžeme přeložit na nejvýše dva dotazy na lokální struktury a nejvýše jeden dotaz na globální strukturu. Všechny struktury přitom vydají odpověď v konstantním čase.

Procedura SoučetÚseku(i, j)
Vstup: Začátek úseku i, konec úseku j

1. Pokud $j < i$, úsek je prázdný, takže položíme $s \leftarrow 0$ a skončíme.
2. $z \leftarrow \lfloor i/b \rfloor$, $k \leftarrow \lfloor j/b \rfloor$ ◁ *ve kterém bloku úsek začíná a kde končí*
3. Pokud $z = k$: ◁ *celý úsek leží v jednom bloku*
4. $\quad s \leftarrow$ LokálníDotaz$(L_z, i \bmod b, j \bmod b)$
5. Jinak:
6. $\quad s_1 \leftarrow$ LokálníDotaz$(L_z, i \bmod b, b - 1)$
7. $\quad s_2 \leftarrow$ GlobálníDotaz$(G, z + 1, k - 1)$
8. $\quad s_3 \leftarrow$ LokálníDotaz$(L_k, 0, j \bmod b)$
9. $\quad s \leftarrow s_1 + s_2 + s_3$

Výstup: Součet úseku s

Nyní se podívejme, co způsobí změna jednoho prvku posloupnosti. Především musíme aktualizovat příslušnou lokální strukturu, což trvá $\Theta(b)$. Pak změnit jeden součet bloku a přepočítat globální strukturu. To zabere čas $\Theta(n/b)$.

Celkem nás tedy změna prvku stojí $\Theta(b+n/b)$. Využijeme toho, že parametr b jsme si mohli zvolit jakkoliv, takže ho nastavíme tak, abychom výraz $b+n/b$ minimalizovali. S rostoucím b první člen roste a druhý klesá. Jelikož součet se asymptoticky chová stejně jako maximum, výraz bude nejmenší, pokud se b a n/b vyrovnají. Zvolíme tedy $b = \lfloor\sqrt{n}\rfloor$ a dostaneme časovou složitost $\Theta(\sqrt{n})$.

Věta: Bloková struktura pro součty úseků se inicializuje v čase $\Theta(n)$, na dotazy odpovídá v čase $\Theta(1)$ a po změně jednoho prvku ji lze aktualizovat v čase $\Theta(\sqrt{n})$.

Odmocninový čas na změnu není optimální, ale princip rozkladu na bloky je užitečné znát a v příštím oddílu nás dovede k mnohem rychlejší struktuře.

Intervalová minima

Pokud se místo součtů budeme ptát na minima úseků, překvapivě dostaneme velmi odlišný problém. Pokusíme-li se použít osvědčený trik a předpočítat prefixová minima, tvrdě narazíme: minimum obecného úseku nelze získat z prefixových minim – například v posloupnosti $\{1, 9, 4, 7, 2\}$ jsou všechna prefixová minima rovna 1.

Opět nám pomůže rozklad na bloky. Lokální struktury si nebudou nic předpočítávat a dotazy budou vyřizovat otrockým projitím celého bloku v čase $\Theta(b)$. Globální struktura si bude pamatovat pouze n/b minim jednotlivých bloků, dotazy bude vyřizovat též otrocky v čase $\Theta(n/b)$.

Inicializaci struktury evidentně zvládneme v lineárním čase. Libovolný dotaz rozdělíme na konstantně mnoho dotazů na lokální a globální struktury, což dohromady potrvá $\Theta(b + n/b)$. Po změně prvku stačí přepočítat minimum jednoho bloku v čase $\Theta(b)$. Použijeme-li osvědčenou volbu $b = \sqrt{n}$, dosáhneme složitosti $\Theta(\sqrt{n})$ pro dotazy i modifikace.

Pro zvědavého čtenáře dodáváme, že existuje i statická struktura s lineárním časem na předvýpočet a konstantním na minimový dotaz. Je ovšem o něco obtížnější, takže zájemce odkazujeme na kapitolu o dekompozici stromů v knize Krajinou grafových algoritmů [**?**]. Jednu z technik, které se k tomu hodí, si můžete vyzkoušet v cvičení 12.

Cvičení

1. Vyřešte úlohu o úseku s maximálním součtem z úvodní kapitoly pomocí prefixových součtů.
2. Je dána posloupnost přirozených čísel a číslo s. Chceme zjistit, zda existuje úsek posloupnosti, jehož součet je přesně s. Jak se úloha změní, pokud dovolíme i záporná čísla?
3. Vymyslete algoritmus, který v posloupnosti celých čísel najde úsek se součtem co nejbližším danému číslu.
4. V posloupnosti celých čísel najděte nejdelší *vyvážený* úsek, tedy takový, v němž je stejně kladných čísel jako záporných.

5.* Mějme posloupnost červených, zelených a modrých prvků. Opět hledáme nejdelší vyvážený úsek, tedy takový, v němž jsou všechny barvy zastoupeny stejným počtem prvků. Co se změní, je-li barev více?

6. Navrhněte dvojrozměrnou analogii prefixových součtů: Pro matici $m \times n$ předpočítejte v čase $\mathcal{O}(mn)$ údaje, pomocí nichž půjde v konstantním čase vypočíst součet hodnot v libovolné souvislé obdélníkové podmatici.

7.* Jak by vypadaly prefixové součty pro d-rozměrnou matici?

8.* Pro ctitele algebry: Myšlenka skládání úseků z prefixů fungovala pro součty, ale selhala pro minima. Uvažujme tedy obecněji nějakou binární operaci $\oplus$, již chceme vyhodnocovat pro úseky: $x_i \oplus x_{i+1} \oplus \ldots \oplus x_j$. Co musí operace $\oplus$ splňovat, aby bylo možné použít prefixové součty? Jaké vlastnosti jsou potřeba pro blokovou strukturu?

9. K odmocninové časové složitosti aktualizací nám pomohlo zavedení dvojúrovňové struktury. Ukažte, jak pomocí tří úrovní dosáhnout času $\mathcal{O}(n^{1/3})$ na aktualizaci a $\mathcal{O}(1)$ na dotaz.

10.* Vyřešte předchozí cvičení pro obecný počet úrovní. Jaký počet je optimální?

11. Na kraji města stojí n-patrový panelák, jehož obyvatelé se baví házením vajíček na chodník před domem. Ideální vajíčko se při hodu z p-tého nebo vyššího patra rozbije; pokud ho hodíme z nižšího, zůstane v původním stavu. Jak na co nejméně pokusů zjistit, kolik je p, pokud máme jenom 2 vajíčka? Jak to dopadne pro neomezený počet vajíček? A jak pro 3?

12. Jak náročný předvýpočet je potřeba, abychom uměli minima úseků počítat v konstantním čase? V čase $\mathcal{O}(n^2)$ je to triviální, ukažte, že stačí $\mathcal{O}(n \log n)$. Hodí se přepočítat minima úseků tvaru $x_i, \ldots, x_{i+2^j-1}$ pro všechna i a j.

13.* V matici tvaru $R \times S$ najděte podmatici tvaru $r \times s$, jejíž medián je největší možný.

1.5. Intervalové stromy

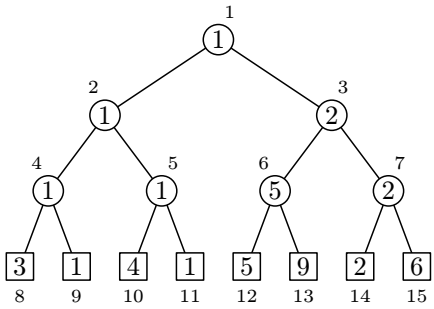
Rozklad posloupnosti na bloky, který jsme zavedli v minulém oddílu, lze elegantně zobecnit. Posloupnost budeme dělit na poloviny, ty zase na poloviny, a tak dále, až dojdeme k jednotlivým prvkům. Pro každou část tohoto rozkladu si přitom budeme udržovat něco předpočítaného.

Tato úvaha vede k takzvaným intervalovým stromům, které dovedou v logaritmickém čase jak vyhodnocovat intervalové dotazy, tak měnit prvky. Nadefinujeme je pro výpočet minim, ale pochopitelně by mohly počítat i součty či jiné operace.

Značení: V celém oddílu pracujeme s posloupností $x_0, \ldots, x_{n-1}$ celých čísel. Bez újmy na obecnosti budeme předpokládat, že n je mocnina dvojky. Interval $\langle i, j \rangle$ obsahuje prvky $x_i, \ldots, x_{j-1}$ (pozor, x_j už v intervalu neleží!). Pro $i \geq j$ to je prázdný interval.

Definice: *Intervalový strom* pro posloupnost $x_0, \ldots, x_{n-1}$ je binární strom s následujícími vlastnostmi:

1. Všechny listy leží na stejné hladině a obsahují zleva doprava prvky $x_0, \ldots, x_{n-1}$.
2. Každý vnitřní vrchol má dva syny a pamatuje si minimum ze všech listů ležících pod ním.



Obr. 1.6: Intervalový strom a jeho očíslování

Pozorování: Stejně jako haldu, i intervalový strom můžeme uložit do pole po hladinách. Na indexech 1 až $n-1$ budou ležet vnitřní vrcholy, na indexech n až $2n-1$ listy s prvky x_0 až x_{n-1}. Strom budeme reprezentovat polem S, jehož prvky budou buď členy posloupnosti nebo minima podstromů.

Ještě si všimneme, že podstromy přirozeně odpovídají intervalům v posloupnosti. Pod kořenem leží celá posloupnost, pod syny kořene poloviny posloupnosti, pod jejich syny čtvrtiny, atd. Obecně očíslujeme-li hladiny od 0 (kořen) po $h = \log n$ (listy), bude na k-té hladině ležet 2^k vrcholů. Ty odpovídají *kanonickým intervalům* tvaru $\langle i, i + 2^{h-k} \rangle$ pro i dělitelné 2^{h-k}.

Statický intervalový strom můžeme vytvořit v lineárním čase: zadanou posloupnost zkopírujeme do listů a pak zdola nahoru přepočítáváme minima ve vnitřních vrcholech: každý vnitřní vrchol obdrží minimum z hodnot svých synů. Celkem tím strávíme čas $\mathcal{O}(n + n/2 + n/4 + \ldots + 1) = \mathcal{O}(n)$.

Intervalový dotaz a jeho rozklad

Nyní se zabývejme vyhodnocováním dotazu na minimum intervalu. Zadaný interval rozdělíme na $\mathcal{O}(\log n)$ disjunktních kanonických intervalů. Jejich minima si strom pamatuje, takže stačí vydat jako výsledek minimum z těchto minim.

Příslušné kanonické intervaly můžeme najít třeba rekurzivním prohledáním stromu. Začneme v kořeni. Kdykoliv stojíme v nějakém vrcholu, podíváme se, v jakém vztahu je hledaný interval $\langle i, j \rangle$ s kanonickým intervalem $\langle a, b \rangle$ přiřazeným aktuálnímu vrcholu. Mohou nastat čtyři možnosti:

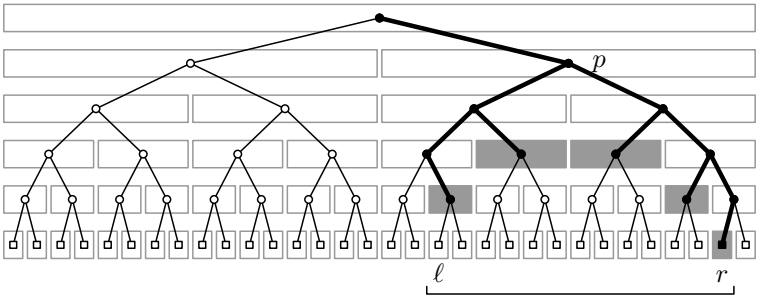
- $\langle i, j \rangle$ a $\langle a, b \rangle$ se shodují: $\langle i, j \rangle$ je kanonický, takže jsme hotovi.
- $\langle i, j \rangle$ leží celý v levé polovině $\langle a, b \rangle$: tehdy se rekurzivně zavoláme na levý podstrom a hledáme v něm stejný interval $\langle i, j \rangle$.
- $\langle i, j \rangle$ leží celý v pravé polovině $\langle a, b \rangle$: obdobně, ale jdeme doprava.
- $\langle i, j \rangle$ prochází přes střed s intervalu $\langle a, b \rangle$: dotaz $\langle i, j \rangle$ rozdělíme na $\langle i, s \rangle$ a $\langle s, j \rangle$. První z nich vyhodnotíme rekurzivně v levém podstromu, druhý v pravém.

Nyní tuto myšlenku zapíšeme v pseudokódu. Na vrcholy se budeme odkazovat pomocí jejich indexů v poli S. Chceme-li rozložit na kanonické intervaly daný interval $\langle i, j\rangle$, zavoláme IntCanon$(1, \langle 0, n\rangle, \langle i, j\rangle)$.

Procedura IntCanon$(v, \langle a, b\rangle, \langle i, j\rangle)$ (rozklad na kanonické intervaly)
Vstup: Index vrcholu v, který odpovídá intervalu $\langle a, b\rangle$; dotaz $\langle i, j\rangle$

1. Pokud $a = i$ a $b = j$, nahlásíme vrchol v a skončíme. ◁ *přesná shoda*
2. $s \leftarrow (a + b)/2$ ◁ *střed intervalu* $\langle a, b\rangle$
3. Pokud $j \leq s$, zavoláme IntCanon$(2v, \langle a, s\rangle, \langle i, j\rangle)$. ◁ *vlevo*
4. Jinak je-li $i \geq s$, zavoláme IntCanon$(2v + 1, \langle s, b\rangle, \langle i, j\rangle)$. ◁ *vpravo*
5. Ve všech ostatních případech: ◁ *dotaz přes střed*
6. Zavoláme IntCanon$(2v, \langle a, s\rangle, \langle i, s\rangle)$.
7. Zavoláme IntCanon$(2v + 1, \langle s, b\rangle, \langle s, j\rangle)$.

Výstup: Rozklad intervalu $\langle i, j\rangle$ na kanonické intervaly



Obr. 1.7: Rozklad dotazu na kanonické intervaly

Lemma: Procedura IntCanon rozloží dotaz na nejvýše $2 \log_2 n$ disjunktních kanonických intervalů a stráví tím čas $\Theta(\log n)$.

Důkaz: Situaci sledujme na obrázku 1.7. Nechť dostaneme dotaz $\langle i, j\rangle$. Označme ℓ a r první a poslední list ležící v tomto intervalu (tyto listy odpovídají prvkům x_i a x_{j-1}). Nechť p je nejhlubší společný předek listů ℓ a r.

Procedura prochází od kořene po cestě do p, protože do té doby platí, že dotaz leží buďto celý nalevo, nebo celý napravo. Ve vrcholu p se dotaz rozdělí na dva podintervaly.

Levý podinterval zpracováváme cestou z p do ℓ. Kdykoliv cesta odbočuje doleva, pravý syn leží celý uvnitř dotazu. Kdykoliv odbočuje doprava, levý syn leží celý venku. Takto dojdeme buďto až do ℓ, nebo dříve zjistíme, že podinterval je kanonický. Na každé z $\log n$ hladin různých od kořene přitom strávíme konstantní čas a vybereme nejvýše jeden kanonický interval.

Pravý podinterval zpracováváme analogicky cestou z p do r. Sečtením přes všechny hladiny získáme kýžené tvrzení. □

Rozklad zdola nahoru

Ukážeme ještě jeden způsob, jak dotaz rozkládat na kanonické intervaly. Tentokrát bude založen na procházení hladin stromu od nejnižší k nejvyšší. Dotaz přitom budeme postupně zmenšovat „ukusováním“ kanonických intervalů z jednoho či druhého okraje. V každém okamžiku výpočtu si budeme pamatovat souvislý úsek vrcholů $\langle a, b\rangle$ na aktuální hladině, které dohromady pokrývají aktuální dotaz.

Na počátku dostaneme dotaz $\langle i, j\rangle$ a přeložíme ho na úsek listů $\langle n+i, n+j\rangle$. Kdykoliv pak na nějaké hladině zpracováváme úsek $\langle a, b\rangle$, nejprve se podíváme, zda jsou a i b sudá. Pokud ano, úsek $\langle a, b\rangle$ pokrývá stejný interval, jako úsek $\langle a/2, b/2\rangle$ o hladinu výš, takže se můžeme na vyšší hladinu rovnou přesunout. Je-li a liché, ukousneme kanonický interval vrcholu a a zbude nám úsek $\langle a+1, b\rangle$. Podobně je-li b liché, ukousneme interval vrcholu $b-1$ a snížíme b o 1. Takto umíme všechny případy převést na sudé a i b, a tím pádem na úsek o hladinu výš.

Zastavíme se v okamžiku, kdy dostaneme prázdný úsek, což je nejpozději tehdy, když se pokusíme vystoupit z kořene nahoru.

Procedura IntCanon2(i, j) (rozklad na kanonické intervaly zdola nahoru)
Vstup: Dotaz $\langle i, j\rangle$

1. $a \leftarrow i+n$, $b \leftarrow j+n$ ◁ *indexy listů*
2. Dokud $a < b$:
3. Je-li a liché, nahlásíme vrchol a a položíme $a \leftarrow a+1$.
4. Je-li b liché, položíme $b \leftarrow b-1$ a nahlásíme vrchol b.
5. $a \leftarrow a/2$, $b \leftarrow b/2$

Výstup: Rozklad intervalu $\langle i, j\rangle$ na kanonické intervaly

Během výpočtu projdeme $\log_2 n+1$ hladin, na každé nahlásíme nejvýše 2 vrcholy. Pokud si navíc uvědomíme, že nahlášení kořene vylučuje nahlášení kteréhokoliv jiného vrcholu, vyjde nám opět nejvýše $2\log_2 n$ kanonických intervalů.

Aktualizace prvku

Od statického intervalového stromu je jen krůček k dynamickému. Co se stane, změníme-li nějaký prvek posloupnosti? Upravíme hodnotu v příslušném listu stromu a pak musíme přepočítat všechny kanonické intervaly, v nichž daný prvek leží. Ty odpovídají vrcholům ležícím na cestě z upraveného listu do kořene stromu.

Stačí tedy změnit list, vystoupat z něj do kořene a cestou všem vnitřním vrcholům přepočítat hodnotu jako minimum ze synů. To stihneme v čase $\Theta(\log n)$. Program je přímočarý:

Procedura IntUpdate(i, x) (aktualizace prvku v intervalovém stromu)
Vstup: Pozice i v posloupnosti, nová hodnota x

1. $a \leftarrow n+i$ ◁ *index listu*

2. $S[a] \leftarrow x$
3. Dokud $a > 1$:
4. $\quad a \leftarrow \lfloor a/2 \rfloor$
5. $\quad S[a] \leftarrow \min(S[2a], S[2a+1])$

Aktualizace intervalu a líné vyhodnocování*

Nejen dotazy, ale i aktualizace mohou pracovat s intervalem. Naučíme náš strom pro výpočet minim operaci IncRange(i, j, δ), která ke všem prvkům v intervalu $\langle i, j\rangle$ přičte δ. Nemůžeme to samozřejmě udělat přímo – to by trvalo příliš dlouho. Použijeme proto trik, kterému se říká *líné vyhodnocování operací.*

Zadaný interval $\langle i, j\rangle$ nejprve rozložíme na kanonické intervaly. Pro každý z nich pak prostě zapíšeme do příslušného vrcholu stromu instrukci „někdy později zvyš všechny hodnoty v tomto podstromu o δ".

Až později nějaká další operace na instrukci narazí, pokusí se ji vykonat. Udělá to ovšem líně: Místo aby pracně zvýšila všechny hodnoty v podstromu, jenom předá svou instrukci oběma svým synům, aby ji časem vykonali. Pokud budeme strom procházet vždy shora dolů, bude platit, že v části stromu, do níž jsme se dostali, jsou už všechny instrukce provedené. Zkratka a dobře, šikovný šéf všechnu práci předává svým podřízeným.

Budeme si proto pro každý vrchol v pamatovat nejen minimum $S[v]$, ale také nějaké číslo $\Delta[v]$, o které mají být zvětšené všechny hodnoty v podstromu: jak prvky v listech, tak všechna minima ve vnitřních vrcholech.

Proceduru pro operaci IncRange odvodíme od průchodu shora dolů v proceduře IntCanon. Místo hlášení kanonických intervalů do nich budeme rozmisťovat instrukce. Navíc potřebujeme aktualizovat minima ve vrcholech ležících mezi kořenem a těmito kanonickými intervaly. To snadno zařídíme při návratech z rekurze. Pro zvýšení intervalu $\langle i, j\rangle$ o δ budeme volat IntIncRange$(1, \langle 0, n\rangle, \langle i, j\rangle, \delta)$.

Procedura IntIncRange$(v, \langle a, b\rangle, \langle i, j\rangle, \delta)$ (aktualizace intervalu)
Vstup: Stojíme ve vrcholu v pro interval $\langle a, b\rangle$ a přičítáme δ k $\langle i, j\rangle$

1. Pokud $a = i$ a $b = j$: ◁ *už máme kanonický interval*
2. $\quad$ Položíme $\Delta[v] \leftarrow \Delta[v] + \delta$ a skončíme.
3. $s \leftarrow (a + b)/2$ ◁ *střed intervalu* $\langle a, b\rangle$
4. Pokud $j \leq s$, zavoláme IntIncRange$(2v, \langle a, s\rangle, \langle i, j\rangle, \delta)$.
5. Jinak je-li $i \geq s$, zavoláme IntIncRange$(2v+1, \langle s, b\rangle, \langle i, j\rangle, \delta)$.
6. Ve všech ostatních případech: ◁ *dotaz přes střed*
7. $\quad$ Zavoláme IntIncRange$(2v, \langle a, s\rangle, \langle i, s\rangle, \delta)$.
8. $\quad$ Zavoláme IntIncRange$(2v+1, \langle s, b\rangle, \langle s, j\rangle, \delta)$.
9. $S[v] \leftarrow \min(S[2v] + \Delta[2v], S[2v+1] + \Delta[2v+1])$

Procedura běží v čase $\Theta(\log n)$, protože projde tutéž část stromu jako procedura IntCanon a v každém vrcholu stráví konstantní čas.

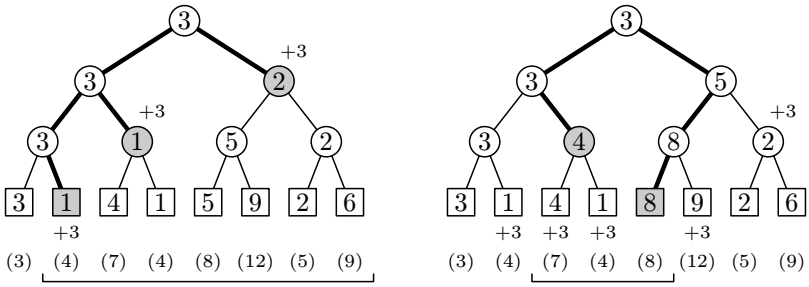
Všechny ostatní operace odvodíme z procházení shora dolů a upravíme je tak, aby v každém navštíveném vrcholu volaly následující proceduru. Ta se postará o líné vyhodnocování instrukcí a zabezpečí, aby v navštívené části stromu žádné instrukce nezbývaly.

Procedura IntLazyEval(v) (líné vyhodnocování)
Vstup: Index vrcholu v

1. $\delta \leftarrow \Delta[v]$, $\Delta[v] \leftarrow 0$
2. $S[v] \leftarrow S[v] + \delta$
3. Pokud $v < n$: ◁ *předáváme synům*
4. $\quad \Delta[2v] \leftarrow \Delta[2v] + \delta$
5. $\quad \Delta[2v+1] \leftarrow \Delta[2v+1] + \delta$

Každou operaci jsme opět zpomalili konstanta-krát, takže jsme zachovali složitost $\Theta(\log n)$.

Vše si můžete prohlédnout na obrázku 1.8. Začneme stromem z obrázku 1.6. Pak přičteme 3 k intervalu $\langle 1, 8\rangle$ a získáme levý strom (u vrcholů jsou napsané instrukce $\Delta[v]$, v závorkách pod listy skutečné hodnoty posloupnosti). Nakonec položíme dotaz $\langle 2, 5\rangle$, čímž se instrukce částečně vyhodnotí a vznikne pravý strom.



Obr. 1.8: Líné vyhodnocování operací

Cvičení

1. Naučte intervalový strom zjistit *druhý nejmenší* prvek v zadaném intervalu.
2. Naučte intervalový strom zjistit nejbližší prvek, který leží napravo od zadaného listu a obsahuje větší hodnotu.
3. Sestrojte variantu intervalového stromu, v níž hranice intervalů nebudou čísla $1, \ldots, n$, ale prvky nějaké obecné posloupnosti $h_1 < \ldots < h_n$.
4. Naprogramujte funkci IntCanon nerekurzivně. Nejprve se vydejte z kořene do společného předka p a pak paralelně procházejte levou i pravou cestu do krajů intervalu. Může se hodit, že bratr vrcholu v má index v xor 1.
5. Jeřáb se skládá z n ramen spojených klouby. Pro jednoduchost si ho představíme jako lomenou čáru v rovině. První úsečka je fixní, každá další je připojena

kloubem na svou předchůdkyni. Koncový bod poslední úsečky hraje roli háku. Navrhněte datovou strukturu, která si bude pamatovat stav jeřábu a bude nabízet operace „otoč i-tým kloubem o úhel α“ a „zjisti aktuální pozici háku“.

6. Ukládáme-li intervalový strom do pole, potřebujeme předem vědět, jak velké pole si pořídit. Ukažte, jak se bez tohoto předpokladu obejít. Může se hodit technika nafukovacího pole z oddílu **??**.

7.* Naučte intervalový strom operaci SetRange(i, j, x), která všechny prvky v intervalu $\langle i, j\rangle$ nastaví na x. Líným vyhodnocováním dosáhněte složitosti $\mathcal{O}(\log n)$.

8.* Vraťte se k cvičení 1.4.10 a všimněte si, že je-li n mocnina dvojky a zvolíte-li počet úrovní rovný $\log_2 n$, stane se z přihrádkové struktury intervalový strom.

9.* Navrhněte datovou strukturu, která si bude pamatovat posloupnost n levých a pravých závorek a bude umět v čase $\mathcal{O}(\log n)$ otočit jednu závorku a rozhodnout, zda je zrovna posloupnost správně uzávorkovaná.