Anotace

- Červeno-černé stromy,
- A-B-stromy, B-stromy.
- Řídké polynomy a matice,
- údržba paměti svépomocí,
- hashování,
- haldy,
- dynamické programování.

Červeno-černé stromy

- Jiná metoda jak udržet strom dostatečně košatý.

Červeno-černé stromy

- Jiná metoda jak udržet strom dostatečně košatý.
- Každému vrcholu přidělíme červenou nebo černou barvu.

Červeno-černé stromy

- Jiná metoda jak udržet strom dostatečně košatý.
- Každému vrcholu přidělíme červenou nebo černou barvu.
- Červené vrcholy nesmějí být dva za sebou,

Červeno-černé stromy

- Jiná metoda jak udržet strom dostatečně košatý.
- Každému vrcholu přidělíme červenou nebo černou barvu.
- Červené vrcholy nesmějí být dva za sebou,
- černých musí být na cestách z každého vrcholu do všech jeho listů stejně.

Červeno-černé stromy

- Jiná metoda jak udržet strom dostatečně košatý.
- Každému vrcholu přidělíme červenou nebo černou barvu.
- Červené vrcholy nesmějí být dva za sebou,
- černých musí být na cestách z každého vrcholu do všech jeho listů stejně.
- Tudíž jeden podstrom musí mít hloubku nejvýše dvakrát větší než druhý.

Červeno-černé stromy

- Jiná metoda jak udržet strom dostatečně košatý.
- Každému vrcholu přidělíme červenou nebo černou barvu.
- Červené vrcholy nesmějí být dva za sebou,
- černých musí být na cestách z každého vrcholu do všech jeho listů stejně.
- Tudíž jeden podstrom musí mít hloubku nejvýše dvakrát větší než druhý.
- K údržbě se používají rotace, dvojité rotace a přebarvování.

Červeno-černé stromy

- Jiná metoda jak udržet strom dostatečně košatý.
- Každému vrcholu přidělíme červenou nebo černou barvu.
- Červené vrcholy nesmějí být dva za sebou,
- černých musí být na cestách z každého vrcholu do všech jeho listů stejně.
- Tudíž jeden podstrom musí mít hloubku nejvýše dvakrát větší než druhý.
- K údržbě se používají rotace, dvojité rotace a přebarvování.
- Pravidla jsou velmi komplikovaná.

Červeno-černé stromy

- Jiná metoda jak udržet strom dostatečně košatý.
- Každému vrcholu přidělíme červenou nebo černou barvu.
- Červené vrcholy nesmějí být dva za sebou,
- černých musí být na cestách z každého vrcholu do všech jeho listů stejně.
- Tudíž jeden podstrom musí mít hloubku nejvýše dvakrát větší než druhý.
- K údržbě se používají rotace, dvojité rotace a přebarvování.
- Pravidla jsou velmi komplikovaná.
- Hloubka červeno-černého stromu je též logaritmická vůči počtu vrcholů.

A-B Stromy

úplně jiný systém správy stromu

- A-B-strom není binární, ale B-ární (tedy každý vrchol má nejvýš B synů)

A-B Stromy

úplně jiný systém správy stromu

- A-B-strom není binární, ale B-ární (tedy každý vrchol má nejvýš B synů)
- V A-B-stromu je každý vrchol buďto list, nebo kořen, nebo má alespoň A a nejvýše B synů. List, který není kořen, obsahuje aspoň $A - 1$ a nejvýše $B - 1$ hodnot.

A-B Stromy

úplně jiný systém správy stromu

- A-B-strom není binární, ale B-ární (tedy každý vrchol má nejvýš B synů)
- V A-B-stromu je každý vrchol buďto list, nebo kořen, nebo má alespoň A a nejvýše B synů. List, který není kořen, obsahuje aspoň $A - 1$ a nejvýše $B - 1$ hodnot.
- Parametry A a B jsou přirozená čísla. Aby bylo možno A-B-strom postavit, je třeba, aby $B \geq 2A - 1$.

A-B Stromy

úplně jiný systém správy stromu

- A-B-strom není binární, ale B-ární (tedy každý vrchol má nejvýš B synů)
- V A-B-stromu je každý vrchol buďto list, nebo kořen, nebo má alespoň A a nejvýše B synů. List, který není kořen, obsahuje aspoň $A - 1$ a nejvýše $B - 1$ hodnot.
- Parametry A a B jsou přirozená čísla. Aby bylo možno A-B-strom postavit, je třeba, aby $B \geq 2A - 1$.
- V každém vrcholu je klíč vždy mezi dvěma pointery a stále platí vlastnost vyhledávacího stromu.

A-B Stromy

úplně jiný systém správy stromu

- A-B-strom není binární, ale B-ární (tedy každý vrchol má nejvýš B synů)
- V A-B-stromu je každý vrchol buďto list, nebo kořen, nebo má alespoň A a nejvýše B synů. List, který není kořen, obsahuje aspoň $A-1$ a nejvýše $B-1$ hodnot.
- Parametry A a B jsou přirozená čísla. Aby bylo možno A-B-strom postavit, je třeba, aby $B \geq 2A-1$.
- V každém vrcholu je klíč vždy mezi dvěma pointery a stále platí vlastnost vyhledávacího stromu.
- A-B-strom obhospodařujeme tak, že je-li vrchol přeplněný, rozštípneme ho na dva a medián pošleme do otce.

A-B Stromy

úplně jiný systém správy stromu

- A-B-strom není binární, ale B-ární (tedy každý vrchol má nejvýš B synů)
- V A-B-stromu je každý vrchol buďto list, nebo kořen, nebo má alespoň A a nejvýše B synů. List, který není kořen, obsahuje aspoň $A-1$ a nejvýše $B-1$ hodnot.
- Parametry A a B jsou přirozená čísla. Aby bylo možno A-B-strom postavit, je třeba, aby $B \geq 2A-1$.
- V každém vrcholu je klíč vždy mezi dvěma pointery a stále platí vlastnost vyhledávacího stromu.
- A-B-strom obhospodařujeme tak, že je-li vrchol přeplněný, rozštípneme ho na dva a medián pošleme do otce.
- Tímto přesunem "do otce" můžeme spustit kaskádu.

Vkládání a ubírání z A-B-stromu

- A-B-strom je vyhledávací strom, tudíž poloha prvku je určena hodnotami prvků v rodičích.
- Při vkládání najdeme list, do kterého by prvek patřil. Pokud se prvek vejde, přidáme ho. Pokud ne, vrchol rozdělíme na dva a medián převedeme do rodiče (jako pivot).
- Štěpení může vyvolat kaskádu vedoucí až ke kořeni (včetně).
- Při rušení prvku problém převedeme na rušení listového vrcholu (ekvivalent "najdi nejlevější v pravém podstromě").
- Pokud vrchol začne být podkritický, buďto si "půjčíme" od bratra, pokud je podkritický i bratr, vrcholy sloučíme (a přidáme pivot z rodiče mezi nimi).

Analýza a poznámky

- Všechny listy jsou ve stejné hloubce.

Analýza a poznámky

- Všechny listy jsou ve stejné hloubce.
- A-B-strom "obsahuje" vyvážený binární strom (tedy lze z něj takový vybrat), proto je hloubka logaritmická vůči počtu prvků.

Analýza a poznámky

- Všechny listy jsou ve stejné hloubce.
- A-B-strom "obsahuje" vyvážený binární strom (tedy lze z něj takový vybrat), proto je hloubka logaritmická vůči počtu prvků.
- Přidáváme i ubíráme v logaritmickém čase.

Analýza a poznámky

- Všechny listy jsou ve stejné hloubce.
- A-B-strom "obsahuje" vyvážený binární strom (tedy lze z něj takový vybrat), proto je hloubka logaritmická vůči počtu prvků.
- Přidáváme i ubíráme v logaritmickém čase.
- Typický příklad je 2-3 strom (každý vrchol obsahuje 1 nebo 2 prvky).

Analýza a poznámky

- Všechny listy jsou ve stejné hloubce.
- A-B-strom "obsahuje" vyvážený binární strom (tedy lze z něj takový vybrat), proto je hloubka logaritmická vůči počtu prvků.
- Přidáváme i ubíráme v logaritmickém čase.
- Typický příklad je 2-3 strom (každý vrchol obsahuje 1 nebo 2 prvky).
- Nepříjemná implementace (je třeba prohledávat více prvků ve vrcholu, evidovat počty prvků ve vrcholu).

Analýza a poznámky

- Všechny listy jsou ve stejné hloubce.
- A-B-strom "obsahuje" vyvážený binární strom (tedy lze z něj takový vybrat), proto je hloubka logaritmická vůči počtu prvků.
- Přidáváme i ubíráme v logaritmickém čase.
- Typický příklad je 2-3 strom (každý vrchol obsahuje 1 nebo 2 prvky).
- Nepříjemná implementace (je třeba prohledávat více prvků ve vrcholu, evidovat počty prvků ve vrcholu).
- A-B-stromy jsou prakticky často používány.

Analýza a poznámky

- Všechny listy jsou ve stejné hloubce.
- A-B-strom "obsahuje" vyvážený binární strom (tedy lze z něj takový vybrat), proto je hloubka logaritmická vůči počtu prvků.
- Přidáváme i ubíráme v logaritmickém čase.
- Typický příklad je 2-3 strom (každý vrchol obsahuje 1 nebo 2 prvky).
- Nepříjemná implementace (je třeba prohledávat více prvků ve vrcholu, evidovat počty prvků ve vrcholu).
- A-B-stromy jsou prakticky často používány.
- Pokud je $A = \lfloor \frac{B+1}{2} \rfloor$, hovoříme o B-stromech.

Analýza a poznámky

- Všechny listy jsou ve stejné hloubce.
- A-B-strom "obsahuje" vyvážený binární strom (tedy lze z něj takový vybrat), proto je hloubka logaritmická vůči počtu prvků.
- Přidáváme i ubíráme v logaritmickém čase.
- Typický příklad je 2-3 strom (každý vrchol obsahuje 1 nebo 2 prvky).
- Nepříjemná implementace (je třeba prohledávat více prvků ve vrcholu, evidovat počty prvků ve vrcholu).
- A-B-stromy jsou prakticky často používány.
- Pokud je $A = \lfloor \frac{B+1}{2} \rfloor$, hovoříme o B-stromech.
- Existují různé modifikace (zvané B+ strom, B*, někdy na listech uděláme spojový seznam, jindy posilujeme "výměny se sourozenci"). Prakticky jsou zajímavé, teoreticky až tolik ne.

A-sort

- A-sort: Třídění pomocí B-stromu s prstem.

A-sort

- A-sort: Třídění pomocí B-stromu s prstem.
- Prst ukazuje na vrchol B-stromu, se kterým jsme pracovali jako s posledním.

A-sort

- A-sort: Třídění pomocí B-stromu s prstem.
- Prst ukazuje na vrchol B-stromu, se kterým jsme pracovali jako s posledním.
- Vkládat nezačínáme od kořene, ale od "prstu".

A-sort

- A-sort: Třídění pomocí B-stromu s prstem.
- Prst ukazuje na vrchol B-stromu, se kterým jsme pracovali jako s posledním.
- Vkládat nezačínáme od kořene, ale od "prstu".
- Dobré výsledky pokud je vstup předtříděný (nebubláme často až do kořene).

Reprezentace spojovými seznamy

- Chceme reprezentovat řídké polynomy (tedy s málo nenulovými koeficienty). Jak to udělat?

Reprezentace spojovými seznamy

- Chceme reprezentovat řídké polynomy (tedy s málo nenulovými koeficienty). Jak to udělat?
- Pomocí spojových seznamů.

Reprezentace spojovými seznamy

- Chceme reprezentovat řídké polynomy (tedy s málo nenulovými koeficienty). Jak to udělat?
- Pomocí spojových seznamů.
- Vhodný je obousměrný spojový seznam...

Reprezentace spojovými seznamy

- Chceme reprezentovat řídké polynomy (tedy s málo nenulovými koeficienty). Jak to udělat?
- Pomocí spojových seznamů.
- Vhodný je obousměrný spojový seznam...
- ...a možná i cyklický – a v tom případě s hlavou!

Reprezentace spojovými seznamy

- Chceme reprezentovat řídké polynomy (tedy s málo nenulovými koeficienty). Jak to udělat?
- Pomocí spojových seznamů.
- Vhodný je obousměrný spojový seznam...
- ...a možná i cyklický – a v tom případě s hlavou!
- Sčítání polynomů: Prolezení spojových seznamů (podobně jako merge).

Reprezentace spojovými seznamy

- Chceme reprezentovat řídké polynomy (tedy s málo nenulovými koeficienty). Jak to udělat?
- Pomocí spojových seznamů.
- Vhodný je obousměrný spojový seznam...
- ...a možná i cyklický – a v tom případě s hlavou!
- Sčítání polynomů: Prolezení spojových seznamů (podobně jako merge).
- Násobení polynomů: Potřebujeme lézt oběma směry.

Reprezentace spojovými seznamy

- Chceme reprezentovat řídké polynomy (tedy s málo nenulovými koeficienty). Jak to udělat?
- Pomocí spojových seznamů.
- Vhodný je obousměrný spojový seznam...
- ...a možná i cyklický – a v tom případě s hlavou!
- Sčítání polynomů: Prolezení spojových seznamů (podobně jako merge).
- Násobení polynomů: Potřebujeme lézt oběma směry.
- Hlava je k tomu, abychom poznali, že jsme na konci.

Reprezentace řídkých matic

- Opět máme málo nenulových prvků, matici tedy komprimujeme.

Reprezentace řídkých matic

- Opět máme málo nenulových prvků, matici tedy komprimujeme.
- Možností je mnoho, je třeba zejména přemýšlet (při použití). Například:

Reprezentace řídkých matic

- Opět máme málo nenulových prvků, matici tedy komprimujeme.
- Možností je mnoho, je třeba zejména přemýšlet (při použití). Například:
- Spojový seznam prvků (uspořádaný v obou dimenzích),

Reprezentace řídkých matic

- Opět máme málo nenulových prvků, matici tedy komprimujeme.
- Možností je mnoho, je třeba zejména přemýšlet (při použití). Například:
- Spojový seznam prvků (uspořádaný v obou dimenzích),
- spojový seznam spojových seznamů (řádky v jednom, sloupce ve druhém),

Reprezentace řídkých matic

- Opět máme málo nenulových prvků, matici tedy komprimujeme.
- Možností je mnoho, je třeba zejména přemýšlet (při použití). Například:
- Spojový seznam prvků (uspořádaný v obou dimenzích),
- spojový seznam spojových seznamů (řádky v jednom, sloupce ve druhém),
- "čtvrcení za živa", tedy rozdělíme matici na čtyři čtvrtiny, ty dělíme dále, až je ve "čtvrtině" málo prvků...

Funkce nízké úrovně

- `procedure mark(var p:pointer);` – oznámí vrchol haldy (kam až bylo allokováno)
  nepoužívat (radí kolega Kryl)

Funkce nízké úrovně

- `procedure mark(var p:pointer);` – oznámí vrchol haldy (kam až bylo allokováno)
  nepoužívat (radí kolega Kryl)
- `procedure release(var p:pointer);` – nastaví vrchol haldy (odallokuje všechno, co je nad vrcholem)
  nepoužívat (DTTO)

Funkce nízké úrovně

- `procedure mark(var p:pointer);` – oznámí vrchol haldy (kam až bylo allokováno)
  nepoužívat (radí kolega Kryl)
- `procedure release(var p:pointer);` – nastaví vrchol haldy (odallokuje všechno, co je nad vrcholem)
  nepoužívat (DTTO)
- Podle všeho ve FPC už nejsou, jsou nebezpečné, je to určitý pokus o garbage collection.

Funkce nízké úrovně

- `procedure mark(var p:pointer);` – oznámí vrchol haldy (kam až bylo allokováno)
  nepoužívat (radí kolega Kryl)
- `procedure release(var p:pointer);` – nastaví vrchol haldy (odallokuje všechno, co je nad vrcholem)
  nepoužívat (DTTO)
- Podle všeho ve FPC už nejsou, jsou nebezpečné, je to určitý pokus o garbage collection.
- `function MemAvail: longint;` – vrátí počet dostupných bytů na haldě
  (není ve Free Pascalu od verze 2.0)

Funkce nízké úrovně

- `procedure mark(var p:pointer);` – oznámí vrchol haldy (kam až bylo allokováno)
  nepoužívat (radí kolega Kryl)
- `procedure release(var p:pointer);` – nastaví vrchol haldy (odallokuje všechno, co je nad vrcholem)
  nepoužívat (DTTO)
- Podle všeho ve FPC už nejsou, jsou nebezpečné, je to určitý pokus o garbage collection.
- `function MemAvail: longint;` – vrátí počet dostupných bytů na haldě
  (není ve Free Pascalu od verze 2.0)
- `function MaxAvail: longint;` – vrátí délku nejdelšího volného bloku (největší naalokovatelnou velikost)
  (DTTO)

Funkce nízké úrovně

- `GetMem` – naalokuje paměť, narozdíl od `new` nezkoumá kolik – používat jen v případě nouze (radí F. Kaempfl)

Funkce nízké úrovně

- `GetMem` – naalokuje paměť, narozdíl od `new` nezkoumá kolik – používat jen v případě nouze (radí F. Kaempfl)
- `FreeMem` – odallokuje paměť naalokovanou pomocí `GetMem` – (DTTO)

Příklad použití GetMem/FreeMem

Vyrobíme pole předem neznámé délky

```
type ppole=^tpole;
     tpole=array[1..10000] of longint;
var pole:ppole;
begin
      GetMem(pole,500);{Urafni 500 bytu}
      pole^[10]:=1000;{To je OK}
      pole^[500]:=1024;{To je K. O., pole je male!}
      FreeMem(pole,500);{Melo by stacit i jen
FreeMem(pole);}
end.
```

Hashování

- Máme-li data, kterými lze indexovat, ale hodnoty by byly příliš velké (například řetězce), má smysl zkusit spočítat nějakou funkci (například posčítat ordinální hodnoty jednotlivých znaků mod 256). Tuto funkci označíme h.

Hashování

- Máme-li data, kterými lze indexovat, ale hodnoty by byly příliš velké (například řetězce), má smysl zkusit spočítat nějakou funkci (například posčítat ordinální hodnoty jednotlivých znaků mod 256). Tuto funkci označíme h.
- Uděláme tak tabulku mnohem menší velikosti, než je universum.

Hashování

- Máme-li data, kterými lze indexovat, ale hodnoty by byly příliš velké (například řetězce), má smysl zkusit spočítat nějakou funkci (například posčítat ordinální hodnoty jednotlivých znaků mod 256). Tuto funkci označíme h.
- Uděláme tak tabulku mnohem menší velikosti, než je universum.
- Tomu říkáme hashování.

Hashování

- Máme-li data, kterými lze indexovat, ale hodnoty by byly příliš velké (například řetězce), má smysl zkusit spočítat nějakou funkci (například posčítat ordinální hodnoty jednotlivých znaků mod 256). Tuto funkci označíme h.
- Uděláme tak tabulku mnohem menší velikosti, než je universum.
- Tomu říkáme hashování.
- Jenže více prvků může kandidovat na totéž místo tabulky, tomu říkáme kolize.

Hashování

řešení kolizí

- Různé metody řešení kolizí:

Hashování

řešení kolizí

- Různé metody řešení kolizí:
- Uděláme spojový seznam,

Hashování

řešení kolizí

- Různé metody řešení kolizí:
- Uděláme spojový seznam,
- srůstající hashování, tedy umísťujeme prvky do přihrádek, kam nepatří,

Hashování

řešení kolizí

- Různé metody řešení kolizí:
- Uděláme spojový seznam,
- srůstající hashování, tedy umísťujeme prvky do přihrádek, kam nepatří,
- z toho plyne problém s implementací `delete` (mazat prakticky nelze, protože bychom mohli roztrhnout řetízek).

Hashování

řešení kolizí

- Různé metody řešení kolizí:
- Uděláme spojový seznam,
- srůstající hashování, tedy umísťujeme prvky do přihrádek, kam nepatří,
- z toho plyne problém s implementací `delete` (mazat prakticky nelze, protože bychom mohli roztrhnout řetízek).
- Na to kam umístit prvek v kolizi existuje mnoho metod. Buďto vybereme další volné místo, nebo obstaráme funkci, která určí další kandidátské místo.

Hashování

řešení kolizí

- Různé metody řešení kolizí:
- Uděláme spojový seznam,
- srůstající hashování, tedy umísťujeme prvky do přihrádek, kam nepatří,
- z toho plyne problém s implementací `delete` (mazat prakticky nelze, protože bychom mohli roztrhnout řetízek).
- Na to kam umístit prvek v kolizi existuje mnoho metod. Buďto vybereme další volné místo, nebo obstaráme funkci, která určí další kandidátské místo.
- Pokud známe množství dat, můžeme se pokusit o perfektní hashování, a to do pole kvadraticky velkého (se středním počtem kolizí menším než 1), nebo dokonce do pole velikosti $4n$, ale pak předem musíme znát data.

Haldy

- Halda je datová struktura umožňující rychle najít minimum.

Haldy

- Halda je datová struktura umožňující rychle najít minimum.
- V haldě platí základní vlastnost, že klíče synů jsou aspoň takové, jako klíč rodiče.

Haldy

- Halda je datová struktura umožňující rychle najít minimum.
- V haldě platí základní vlastnost, že klíče synů jsou aspoň takové, jako klíč rodiče.
- Byla binární halda (za účelem třídění).

Haldy

- Halda je datová struktura umožňující rychle najít minimum.
- V haldě platí základní vlastnost, že klíče synů jsou aspoň takové, jako klíč rodiče.
- Byla binární halda (za účelem třídění).
- Nyní budou další, pozor, pod aliasem strom (halda bude až množina stromů).

Motivace

- Některé algoritmy by rády udržovaly hodnoty tak, aby šlo rychle najít minimum (`extract_min`), případně aby šlo klíč vrcholu snížit (`decrease_key`). Příkladem je Dijkstrův algoritmus.
- Toto bychom mohli zařídit pomocí uspořádaného spojového seznamu.
- Složitost:
  - `extract_min` – $\Theta(1)$,
  - `decrease_key` – $\Theta(n)$.
- `extract_min` je OK, `decrease_key` je K. O. Co jinak?
- Vyhledávací stromy
  - `extract_min` – $O(\log n)$,
  - `decrease_key` – $O(\log n)$.

Binomiální strom

Definition

List je binomiální strom řádu 0.
Binomiální strom je strom řádu k vznikne ze dvou binomiálních stromů řádu $k-1$ tak, že kořen s nižší hodnotou stanovíme kořenem (původní strom řádu $k-1$ okopírujeme nezměněný) a jako poslední syn mu přidáme ten druhý strom řádu $k-1$.
Binomiální halda je množina binomiálních stromů takových, že strom každého řádu se vyskytuje nejvýš jednou.

- Binomiální strom řádu k má 2^k prvků.

Binomiální strom

Definition

List je binomiální strom řádu 0.
Binomiální strom je strom řádu k vznikne ze dvou binomiálních stromů řádu $k-1$ tak, že kořen s nižší hodnotou stanovíme kořenem (původní strom řádu $k-1$ okopírujeme nezměněný) a jako poslední syn mu přidáme ten druhý strom řádu $k-1$.
Binomiální halda je množina binomiálních stromů takových, že strom každého řádu se vyskytuje nejvýš jednou.

- Binomiální strom řádu k má 2^k prvků.
- Pokud uděláme `extract_min`, strom řádu k se rozsype na k stromů (tedy haldu).

Definition

List je binomiální strom řádu 0.
Binomiální strom je strom řádu k vznikne ze dvou binomiálních stromů řádu $k-1$ tak, že kořen s nižší hodnotou stanovíme kořenem (původní strom řádu $k-1$ okopírujeme nezměněný) a jako poslední syn mu přidáme ten druhý strom řádu $k-1$.
Binomiální halda je množina binomiálních stromů takových, že strom každého řádu se vyskytuje nejvýš jednou.

- Pokud jsme měli před `extract_min` haldu a objeví-li se aspoň dva stromy téhož řádu, můžeme z nich (v konstantním čase) postavit strom řádu o jedna větší.

Definition

List je binomiální strom řádu 0.
Binomiální strom je strom řádu k vznikne ze dvou binomiálních stromů řádu $k-1$ tak, že kořen s nižší hodnotou stanovíme kořenem (původní strom řádu $k-1$ okopírujeme nezměněný) a jako poslední syn mu přidáme ten druhý strom řádu $k-1$.
Binomiální halda je množina binomiálních stromů takových, že strom každého řádu se vyskytuje nejvýš jednou.

- Pokud jsme měli před `extract_min` haldu a objeví-li se aspoň dva stromy téhož řádu, můžeme z nich (v konstantním čase) postavit strom řádu o jedna větší.
- Takto můžeme haldu udržovat pro účely `extract_min`, složitost bude $O(\log n)$ (prohledat kořeny, rozlepit "ten správný"strom a zmergovat).

Definition

List je binomiální strom řádu 0.
Binomiální strom je strom řádu k vznikne ze dvou binomiálních stromů řádu $k-1$ tak, že kořen s nižší hodnotou stanovíme kořenem (původní strom řádu $k-1$ okopírujeme nezměněný) a jako poslední syn mu přidáme ten druhý strom řádu $k-1$.
Binomiální halda je množina binomiálních stromů takových, že strom každého řádu se vyskytuje nejvýš jednou.

- Pokud jsme měli před `extract_min` haldu a objeví-li se aspoň dva stromy téhož řádu, můžeme z nich (v konstantním čase) postavit strom řádu o jedna větší.
- Takto můžeme haldu udržovat pro účely `extract_min`, složitost bude $O(\log n)$ (prohledat kořeny, rozlepit "ten správný"strom a zmergovat).
- To je pořád moc, navíc `decrease_key` je prakticky neimplementovatelný.

Krok stranou

Fibonacciho halda

- Fibonacciho strom vznikne podobně jako binomiální (pozor, u kolegy Kryla se Fibonacciho strom říká kritickému AVL-stromu, což je něco úplně jiného a ani s jedním Fibonacci neměl vůbec nic společného!), jenže:

Krok stranou

Fibonacciho halda

- Fibonacciho strom vznikne podobně jako binomiální (pozor, u kolegy Kryla se Fibonacciho strom říká kritickému AVL-stromu, což je něco úplně jiného a ani s jedním Fibonacci neměl vůbec nic společného!), jenže:
- Řádem Fibonacciho stromu je stupeň kořene. Navíc nastavíme každému vrcholu (při přidání) counter na 1 (a udržujeme). Fibonacciho strom řádu k vznikne z Fibonacciho stromu řádu k odříznutím nějakého syna vrcholu s counterem nastaveným na 1 (různého od kořene). Pokud vrcholu odřízneme syna, counter snížíme.

Fibonacciho haldy

- Fibonacciho strom řádu k tudíž má kořen stupně k. Nejvýše jednomu jeho synovi jsme odřízli syna, tedy oproti Binomiálnímu stromu může jeden ze synů mít řád o jedna menší, tedy v nejhorším případě jako bychom Fibonacciho strom postavili ze stromu řádu $k-1$ a $k-2$ (proto Fibonacciho).

Fibonacciho haldy

- Fibonacciho strom řádu k tudíž má kořen stupně k. Nejvýše jednomu jeho synovi jsme odřízli syna, tedy oproti Binomiálnímu stromu může jeden ze synů mít řád o jedna menší, tedy v nejhorším případě jako bychom Fibonacciho strom postavili ze stromu řádu $k-1$ a $k-2$ (proto Fibonacciho).
- Fibonacciho halda je opět množina stromů, kdy strom každého řádu se vyskytuje nejvýše jednou.

Fibonacciho haldy

- Fibonacciho strom řádu k tudíž má kořen stupně k. Nejvýše jednomu jeho synovi jsme odřízli syna, tedy oproti Binomiálnímu stromu může jeden ze synů mít řád o jedna menší, tedy v nejhorším případě jako bychom Fibonacciho strom postavili ze stromu řádu $k-1$ a $k-2$ (proto Fibonacciho).
- Fibonacciho halda je opět množina stromů, kdy strom každého řádu se vyskytuje nejvýše jednou.
- Operace `extract_min` stejně jako pro binomiální haldy.

Fibonacciho halda – konec

- Operace `decrease_key`: Vrcholu sniž hodnotu (klíč).

Fibonacciho halda – konec

- Operace `decrease_key`: Vrcholu sniž hodnotu (klíč).
- Pokud tím dojde k porušení vlastnosti haldy, vrchol odřízni.

Fibonacciho halda – konec

- Operace `decrease_key`: Vrcholu sniž hodnotu (klíč).
- Pokud tím dojde k porušení vlastnosti haldy, vrchol odřízni.
- Pokud měl rodič counter 1, sniž ho (a zařaď nový strom do haldy).

Fibonacciho halda – konec

- Operace `decrease_key`: Vrcholu sniž hodnotu (klíč).
- Pokud tím dojde k porušení vlastnosti haldy, vrchol odřízni.
- Pokud měl rodič counter 1, sniž ho (a zařaď nový strom do haldy).
- Pokud měl rodič counter 0, odřízni ho taky (a prověř jeho rodiče atd.).

Fibonacciho halda – konec

- Operace `decrease_key`: Vrcholu sniž hodnotu (klíč).
- Pokud tím dojde k porušení vlastnosti haldy, vrchol odřízni.
- Pokud měl rodič counter 1, sniž ho (a zařaď nový strom do haldy).
- Pokud měl rodič counter 0, odřízni ho taky (a prověř jeho rodiče atd.).
- Líná implementace: Necháme víc stromů téhož řádu a konsolidujeme jen při `extract_min`.

Fibonacciho halda – konec

- Operace `decrease_key`: Vrcholu sniž hodnotu (klíč).
- Pokud tím dojde k porušení vlastnosti haldy, vrchol odřízni.
- Pokud měl rodič counter 1, sniž ho (a zařaď nový strom do haldy).
- Pokud měl rodič counter 0, odřízni ho taky (a prověř jeho rodiče atd.).
- Líná implementace: Necháme víc stromů téhož řádu a konsolidujeme jen při `extract_min`.
- Výhoda: Amortizovaná složitost `decrease_key` bude konstantní. Funkce `decrease_key` bude sice logaritmická, ale to by byla stejně, nevolá se ale tolikrát (jako `decrease_key`).

Problémy, které byly

- Přednášející jde tentokrát do M1,
- počet platných uzávorkování pomocí n párů závorek,
- počet rozkladů přirozeného čísla na součet nerostoucích kladných celých čísel,
- Pascalův trojúhelník,
- nejdelší rostoucí podposloupnost,
- pořadí násobení matic,
- problém batohu.

Pascalův trojúhelník

- Obsahuje kombinační čísla,

Pascalův trojúhelník

- Obsahuje kombinační čísla,
- n-tý řádek konkrétně obsahuje hodnoty $\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}$,

Pascalův trojúhelník

- Obsahuje kombinační čísla,
- n-tý řádek konkrétně obsahuje hodnoty $\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}$,
- při výpočtu využíváme toho, že $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$,

Pascalův trojúhelník

- Obsahuje kombinační čísla,
- n-tý řádek konkrétně obsahuje hodnoty $\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}$,
- při výpočtu využíváme toho, že $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$,
- pro jednoduchost chceme spočítat jen číslo $\binom{n}{k}$.

Pascalův trojúhelník rekurzívní řešení

- Získali jsme rekurenci $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$,
- z této snadno sestrojíme rekurzívní program:

```
function kombin(n,k:integer):longint;
begin
      if (k=0) or (k=n) then kombin:=1;
      else  kombin:=kombin(n-1,k-1)+kombin(n-1,k);
end;
begin
      kombin(100,50);
end.
```

Mále stejný problém, pořád počítáme to samé mockrát.
Opět přidáme cache na výsledky.

Pascalův trojúhelník

```
const MAX=100;
var cache:array[0..MAX,0..MAX] of longint;
function kom(n,k:integer):longint;
begin
    if cache[n,k]=0 then
    begin if (k=0) or (k=n) then cache[n,k]:=1
       else cache[n,k]:=kom(n-1,k-1)+kom(n-1,k);
    end;
    kom:=cache[n,k];
end;
var n,k:integer;
begin
    read(n,k);
    writeln(kom(n,k));
end
```

Násobení matic

- Násobení matic není komutativní, ale je asociativní.
- Máme-li vynásobit několik matic, nemůžeme jejich pořadí měnit,
- můžeme součin všelijak závorkovat.
- Různá uzávorkování jsou různě výhodná.
- Které z nich je to nejvýhodnější?
- Předpokládáme, že máme abstraktní funkce zjistící ze zadaných rozměrů složitost součinu matic správných tvarů.

Násobení matic

- Některé násobení provedeme jako poslední.

Násobení matic

- Některé násobení provedeme jako poslední.
- To úlohu rozdělí na (nejvýše) dvě další instance.

Násobení matic

- Některé násobení provedeme jako poslední.
- To úlohu rozdělí na (nejvýše) dvě další instance.
- V každé (menší) instanci opět nějaké násobení provedeme jako poslední.

Násobení matic

- Některé násobení provedeme jako poslední.
- To úlohu rozdělí na (nejvýše) dvě další instance.
- V každé (menší) instanci opět nějaké násobení provedeme jako poslední.
- Ideální podhoubí pro rekurzi!

Násobení matic

- function slozitost(od,az_do:integer):longint;
- for i:=od to az_do-1 do begin
- slozitost:=slozitost(od,i)+
  slozitost(i+1,az_do)+samotne_nasobeni;

Opět budeme zbytečně násobit stále to samé.
Opět se toho zbavíme stejně.

Násobení matic

- function slozitost(od,az_do:integer):longint;
- if cache[od,az_do]=0 then
- for i:=od to az_do-1
- cache[od,az_do]:=slozitost(od,i)+ slozitost(i+1,az_do)+samotne_nasobeni;
- slozitost:=cache[od,az_do];

Metoda

- Všechny tyto problémy jsme řešili stejně:

Metoda

- Všechny tyto problémy jsme řešili stejně:
- Navrhli jsme rekurzívní algoritmus,

Metoda

- Všechny tyto problémy jsme řešili stejně:
- Navrhli jsme rekurzívní algoritmus,
- ten počítal často to samé, proto jsme mu doplnili cache.

Metoda

- Všechny tyto problémy jsme řešili stejně:
- Navrhli jsme rekurzívní algoritmus,
- ten počítal často to samé, proto jsme mu doplnili cache.
- Pokud se podíváme pořádně, na správném místě cache najdeme výsledek.

Metoda

- Všechny tyto problémy jsme řešili stejně:
- Navrhli jsme rekurzívní algoritmus,
- ten počítal často to samé, proto jsme mu doplnili cache.
- Pokud se podíváme pořádně, na správném místě cache najdeme výsledek.
- Potřebujeme tam tedy tu rekurzi?

Metoda

- Všechny tyto problémy jsme řešili stejně:
- Navrhli jsme rekurzívní algoritmus,
- ten počítal často to samé, proto jsme mu doplnili cache.
- Pokud se podíváme pořádně, na správném místě cache najdeme výsledek.
- Potřebujeme tam tedy tu rekurzi?
- Ne a můžeme se jí zbavit za předpokladu, že zjistíme, jak vyplňovat cache, tedy tabulku.

Metoda

- Všechny tyto problémy jsme řešili stejně:
- Navrhli jsme rekurzívní algoritmus,
- ten počítal často to samé, proto jsme mu doplnili cache.
- Pokud se podíváme pořádně, na správném místě cache najdeme výsledek.
- Potřebujeme tam tedy tu rekurzi?
- Ne a můžeme se jí zbavit za předpokladu, že zjistíme, jak vyplňovat cache, tedy tabulku.
- Tomu říkáme dynamické programování.

Odstranění rekurze

- Přednášející jde do posluchárny:

```
a[1]:=1;
a[2]:=2;
for i:=3 to n do a[i]:=a[i-1]+a[i-2];
```

- Pascalův trojúhelník:

```
for i:=0 to n do
      for j:=0 to i do
            p(i,j):=p(i-1,j-1)+p(i-1,j);
```

  a okrajové případy zvlášť!

- Závorkování: Cvičení

Násobení matic – poučení

- Tomuto říkáme *dynamické programování.*
- Zpravidla implementujeme pouze fázi vyplňování tabulky.
- Na tom je nepříjemné určovat, odkud vyplňování zahájit.
- Není tudíž špatné navrhnout si aspoň na počátku rekurzívní řešení a až pak rekurzi "rozbít":
- `cache[od,az_do]:=min{cache[od,i]+cache[i,az_do]+ samotne_nasobeni}`
- což stačí udělat cyklem.

Problém batohu

Definition

Problémem batohu nazveme problém, kdy máme zadány váhy jednotlivých předmětů $(w_1, \ldots w_n)$ a jejich ceny $(c_1, \ldots c_n)$ a nosnost batohu m a ptáme se, jak naložit batoh co nejcennějším obsahem.

- Předpokládejme variantu, že všechna čísla jsou přirozená!

Problém batohu

Definition

Problémem batohu nazveme problém, kdy máme zadány váhy jednotlivých předmětů $(w_1, \ldots w_n)$ a jejich ceny $(c_1, \ldots c_n)$ a nosnost batohu m a ptáme se, jak naložit batoh co nejcennějším obsahem.

- Předpokládejme variantu, že všechna čísla jsou přirozená!
- Rekurzivní řešení:

Problém batohu

Definition

Problémem batohu nazveme problém, kdy máme zadány váhy jednotlivých předmětů $(w_1, \ldots w_n)$ a jejich ceny $(c_1, \ldots c_n)$ a nosnost batohu m a ptáme se, jak naložit batoh co nejcennějším obsahem.

- Předpokládejme variantu, že všechna čísla jsou přirozená!
- Rekurzivní řešení:
- Načti předměty a volej funkci přidej(k,l).

Problém batohu

Definition

Problémem batohu nazveme problém, kdy máme zadány váhy jednotlivých předmětů $(w_1, \ldots w_n)$ a jejich ceny $(c_1, \ldots c_n)$ a nosnost batohu m a ptáme se, jak naložit batoh co nejcennějším obsahem.

- Předpokládejme variantu, že všechna čísla jsou přirozená!
- Rekurzivní řešení:
- Načti předměty a volej funkci přidej(k,l).
- První parametr říká, kolik předmětů už v batohu je, druhý určuje index posledního z nich.

Problém batohu

Definition

Problémem batohu nazveme problém, kdy máme zadány váhy jednotlivých předmětů $(w_1, \ldots w_n)$ a jejich ceny $(c_1, \ldots c_n)$ a nosnost batohu m a ptáme se, jak naložit batoh co nejcennějším obsahem.

- Předpokládejme variantu, že všechna čísla jsou přirozená!
- Rekurzivní řešení:
- Načti předměty a volej funkci přidej(k,l).
- První parametr říká, kolik předmětů už v batohu je, druhý určuje index posledního z nich.
- Nevýhoda: Zkoušíme všechny možnosti.

Problém batohu

Definition

Problémem batohu nazveme problém, kdy máme zadány váhy jednotlivých předmětů $(w_1, \ldots w_n)$ a jejich ceny $(c_1, \ldots c_n)$ a nosnost batohu m a ptáme se, jak naložit batoh co nejcennějším obsahem.

- Předpokládejme variantu, že všechna čísla jsou přirozená!
- Rekurzivní řešení:
- Načti předměty a volej funkci přidej(k,l).
- První parametr říká, kolik předmětů už v batohu je, druhý určuje index posledního z nich.
- Nevýhoda: Zkoušíme všechny možnosti.
- Zlepšení? Dynamickým programem, ale neotřelým způsobem.

- Vytvoř pomocné batohy s nosnostmi $1, 2, \ldots m$ a proměnné popisující optimum v nich zinicializuj nulami.
- Pro každý předmět i přepočítej všechny batohy a zjisti, zda dopadnou lépe, pokud předmět přidáme nebo ne:
- Pro batoh k zjisti, pokud `cena_batohu(`k`)<cena_batohu(`$k - w_i$`)` $+ c_i$.
- Pokud ano, `cena_batohu(`k`):=cena_batohu(`$k - w_i$`)` $+ c_i$.
- Invariant: Řešíme-li prvek i, evidujeme (před touto fází) optimální naplnění batohů pomocí prvků $1, 2, \ldots k - 1$.
- Složitost: mn.
- Pozor, pokud nejsou čísla přirozená, stává se problém NP-těžkým. Není ale silně NP-těžký, což znamená, že těžkost se vynucuje "velkými čísly lišícími se o málo".

Konec

Děkuji za pozornost...