Föreläsning 19

Prioritetskö, Heap, Trie, Union/Find, Geometriska tillämpningar av BST

TDDD86: DALP

Utskriftsversion av föreläsning i *Datastrukturer, algoritmer och programmeringsparadigm*
13 november 2015

Tommy Färnqvist, IDA, Linköpings universitet

19.1

Innehåll

Innehåll

19.2

1 Prioritetsköer

Prioritetsköer

En vanligt förekommande situation:

- Väntelista (jobbhantering på fleranvändardatorer, simulering av händelser)
- Om en resurs blir ledig, välj ett element från väntelistan
- Valet är baserat på någon partial/linjär ordning:
  - jobbet med högst prioritet ska köras först,
  - varje händelse ska inträffa vid en viss tidpunkt; händelserna ska bearbetas i tidsordning

19.3

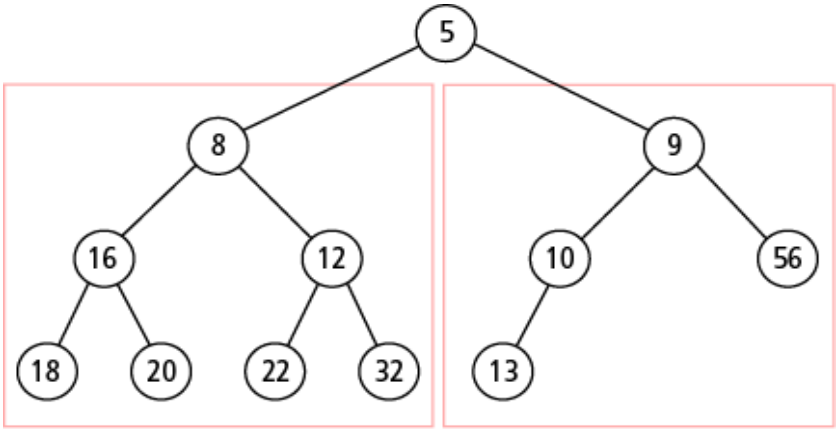
ADT prioritetskö

- Linjärt ordnad mängd K av nycklar
- Vi lagrar par (k, v) (som i ADT Dictionary), *flera par* med *samma nyckel* är tillåtet
- en vanlig operation är att hämta par med minimal nyckel
- Operationer på en prioritetskö P:
  - makeEmptyPQ()
  - isEmpty()
  - size()
  - min(): hitta ett par (k, v) som har minimalt k i P; returnera (k, v)
  - insert(k, v): sätt in (k, v) i P
  - removeMin(): ta bort och returnera ett par (k, v) i P med minimalt k; **error** om P är tom

19.4

Implementation av prioritetsköer

- Vi kan t.ex. använda (sorterade) länkade listor, BST eller Skip-listor
- En annan idé: använd ett fullständigt binärt träd där roten i varje (del)träd T innehåller det minsta elementet i T



Det här är ett partiellt ordnat träd, också kallat heap!

19.5

1.1 Heapar

Att uppdatera en heapstruktur

- Med sista lövet menar vi den sista noden i en traversering i levelorder

- removeMin(PQ) *// ta bort roten*
  - Ersätt roten med sista lövet
  - Återställ den partiella ordningen genom att byta noder nedåt ”down-heap bubbling”

- insert(PQ, k, v)
  - Sätt in ny nod (k, v) efter sista lövet
  - Återställ den partiella ordningen genom ”up-heap bubbling”

19.6

Egenskaper

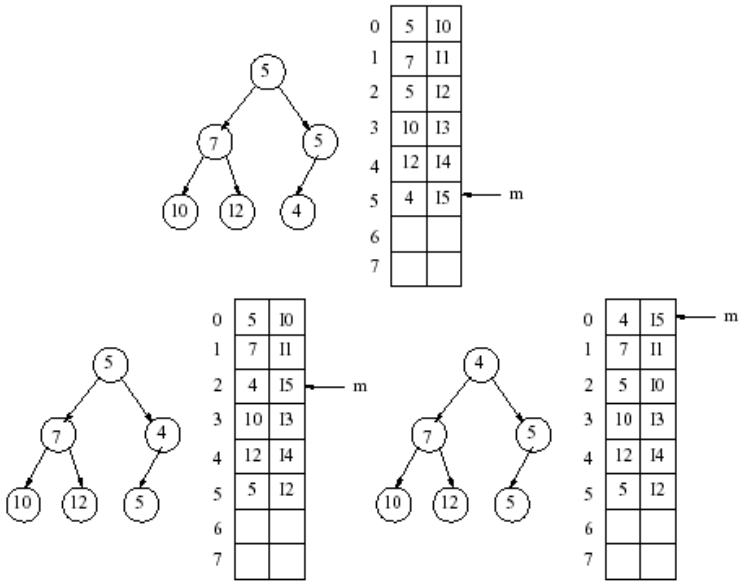
- size(), isEmpty(), min(): $O(1)$
- insert(), removeMin(): $O(\log n)$

Kom ihåg arrayrepresentaionen av BST Ett fullständigt binärt träd...

- Kompakt arrayrepresentation
- ”Bubble-up” och ”bubble-down” har snabba implementationer

19.7

Exempel: ”bubble-up” efter insert(4,15)



19.8

Heapvarianter

Olika partialordningar

- minsta nyckeln i roten (minHeap)
- största nyckeln i roten (maxHeap)

Olika arrayrepresentationer

- numrering framåt i levelorder (med början från 0 eller 1)
- numrering bakåt i levelorder (med början från 0 eller 1)

19.9

1.2 Tillämpning

Giriga algoritmer

Algoritmer som löser en bit av problemet i taget. I varje steg görs det som ger bäst utdelning/kostar minst.

- Den giriga metoden är ett allmänt paradigm för algoritmdesign som bygger på följande:
  - konfigurationer: olika val, samlingar eller värden att hitta
  - målfunktion: konfigurationer tilldelas en poäng som vi vill maximera eller minimera
- Det fungerar bäst applicerat på problem med *greedy-choice*-egenskapen:
  - en globalt optimal lösning kan alltid hittas genom en serie lokala förbättringar utgående från en begynnelse-konfiguration

För många problem ger giriga algoritmer inte optimala lösningar utan kanske hyfsade approximativa lösningar.

19.10

Textkomprimering

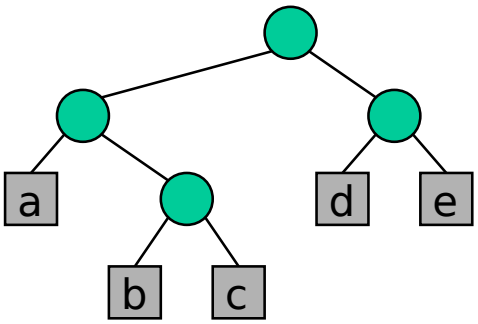
- Givet en sträng X, koda X som en kortare sträng Y
  - Sparar minne/bandbredd
- Ett bra sätt att göra det på: *Huffmankodning*
  - Beräkna frekvensen $f(c)$ för varje tecken c
  - Använd korta koder för tecken med hög frekvens
  - Inget kodord är prefixet av ett annat kodord
  - Använd ett optimalt kodningsträd för att bestämma kodorden

19.11

Exempel på kodningsträd

- En kod avbildar varje tecken i ett alfabet på ett binärt kodord
- En prefixkod är en binär kod sådan att inget kodord är ett prefix av ett annat kodord
- En kodningsträd representerar en prefixkod
  - Varje extern nod lagrar ett tecken
  - Kodordet för ett tecken ges av stigen från roten till den externa noden som lagrar tecknet (0 för ett vänsterbarn och 1 för ett högerbarn)

00	010	011	10	11
a	b	c	d	e



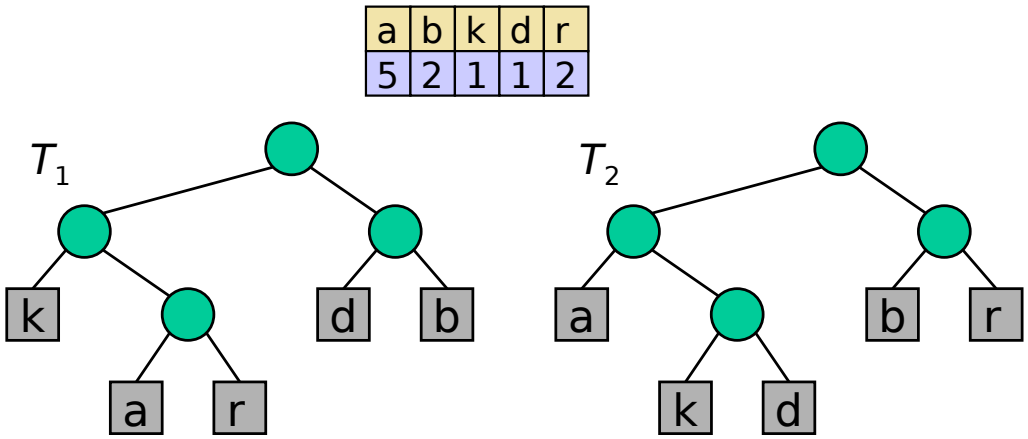
19.12

Optimering av kodningsträd

- Givet en textsträng X vill vi hitta en prefixkod för tecknen i X som ger en liten kodning av X
  - Vanliga tecken borde få korta kodord
  - Ovanliga tecken borde få långa kodord

Exempel: $X =$ *abrakadabra*

- T_1 kodar X med 29 bitar
- T_2 kodar X med 24 bitar



Huffmans algoritm

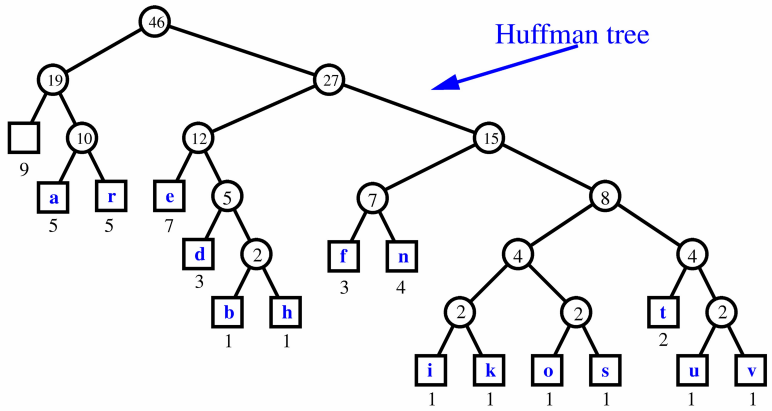
- Givet en sträng X konstruerar Huffmans algoritm en prefixkod som minimerar storleken på kodningen av X
- Algoritmen går i tid $O(n + d \log d)$, där n är storleken på X och d är antalet distinkta tecken i X
- En heapbaserad priokö används som extra datastruktur

function HUFFMANENCODING($X, |X| = n$)
  $C \leftarrow$ DISTINCTCHARACTERS(X)
  COMPUTEFREQUENCIES(C, X)
  $Q \leftarrow$ ny tom heap
  for all $c \in C$ **do**
    $T \leftarrow$ nytt ennodsträd som lagrar c
    Q.INSERT(GETFREQUENCY(C), 1)
  while Q.SIZE() > 1 **do**
    $f_1 \leftarrow Q$.MIN()
    $T_1 \leftarrow Q$.REMOVEMIN()
    $f_2 \leftarrow Q$.MIN()
    $T_2 \leftarrow Q$.REMOVEMIN()
    $T \leftarrow$ JOIN(T_1, T_2)
    Q.INSERT($f_1 + f_2, T$)
  return Q.REMOVEMIN()

Ett till exempel

String: **a fast runner need never be afraid of the dark**

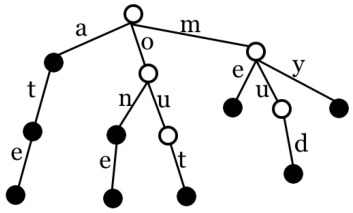
Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



2 Trie

Trie (prefix-träd)

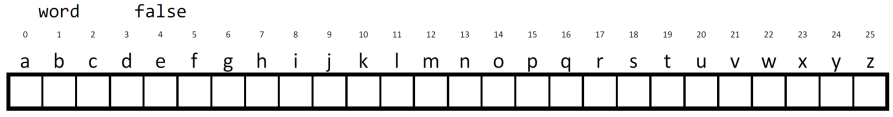
- trie: En ordnad trädstruktur som används för att lagra en mängd data, vanligen strängar, optimerad för att göra prefix-sökningar
  - Exempel: Börjar några ord i mängden med prefixet `mart`?
  - Lexikon-klassen i labb 5 använder en sådan datastruktur
  - Idén: istället för ett binärt träd, använd ett "26-ärt" träd
    - varje nod har 26 barn: en för varje bokstav A-Z
    - lägg till ord i ett trie genom att följa lämpliga barnpekare



Trie-nod

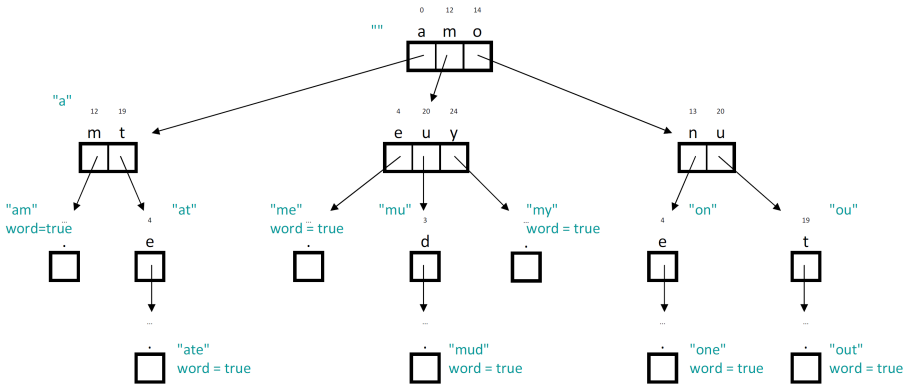
```
struct TrieNode {
   bool word;
   TrieNode* children[26];

   TrieNode() {
      this->word = false;
      for (int i = 0; i < 26; i++) {
         this->children[i] = nullptr;
      }
   }
};
```



Trie med data

- Efter att ha satt in “am”, “ate”, “me”, “mud”, “my”, “one”, “out”:



19.18

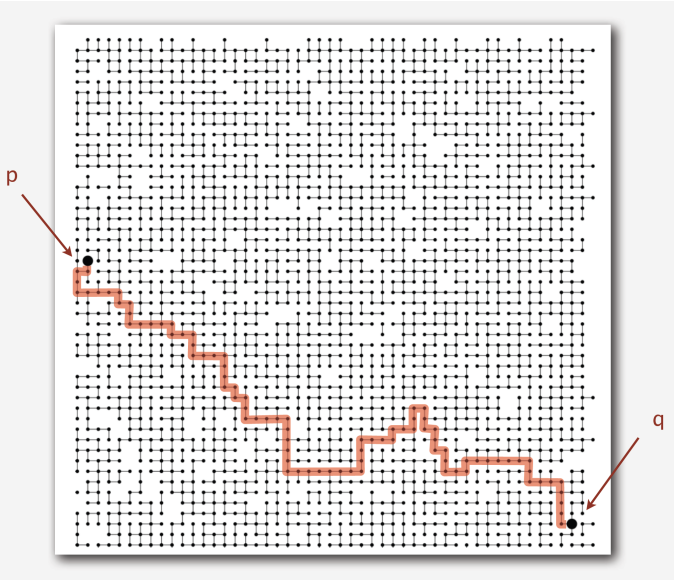
3 Union/Find

Partitioneringar med Union/Find-operationer

- makeSet(x): Skapa en mängd enbart innehållande elementet x och returnera positionen som lagrar x den nya mängden.
- union(A, B): Returnera mängden $A \cup B$, förstör de gamla A och B.
- find(p): Returnera mängden som innehåller elementet i position p.

19.19

Exempel: Dynamisk konnektivitet



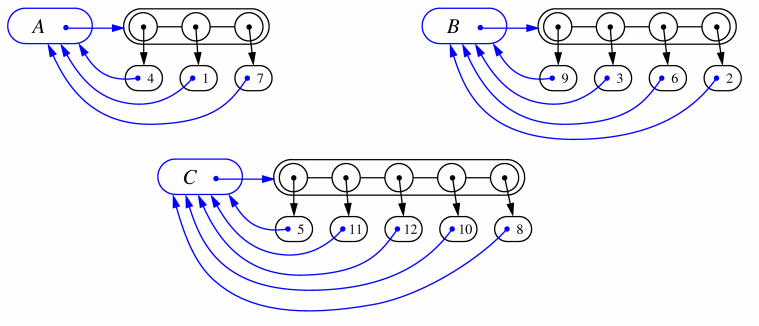
Fråga: finns det en stig mellan p och q?

- Pixlar i ett digital foto
- Datorer i ett nätverk
- Vänner i ett socialt nätverk
- Transistorer på ett datorchip
- Element i en matematisk mängd
- Variabelnamn i ett datorprogram
- Metalliska delar av ett kompositsystem

19.20

Listbaserad implementation

- Varje mängd lagras som en sekvens representerad av en länkad lista
- Varje nod lagrar ett objekt innehållande ett element och en referens till mängdens namn



19.21

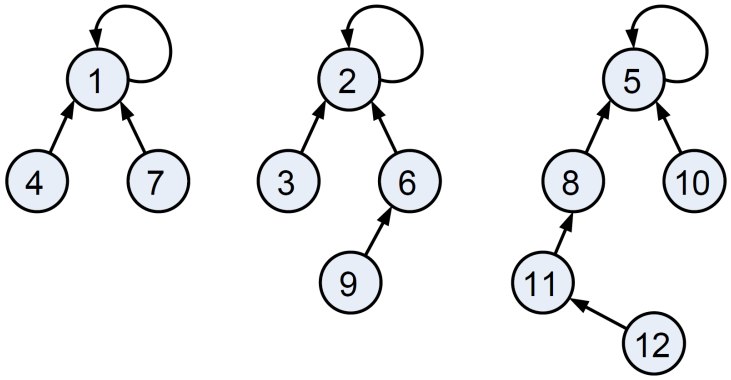
Analys av listbaserad representation

- När union utförs, flytta alltid element från den mindre mängden till den större mängden
  - Varje gång ett element flyttas hamnar det i en mängd som är åtminstone dubbelt så stor som den gamla mängden
  - Alltså, ett element kan flyttas max $O(\log n)$ gånger
- Total tid för att utföra n union- och find-operationer är $O(n \log n)$

19.22

Trädbaserad implementation

- Varje element lagras i en nod som innehåller en pekare till ett mängdnamn
- En nod v vars pekare pekar på nod v är också ett mängdnamn
- Varje mängd är ett träd, rotat i en nod med självrefererande mängdnamnspekare
- T.ex. mängderna ”1”, ”2” och ”5”:

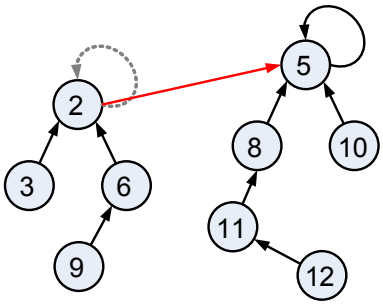


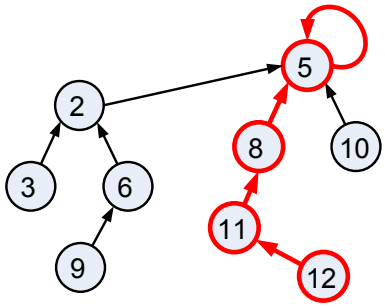
19.23

Operationer

- För att utföra union, låt bara roten av ett träd peka på roten av det andra

- För att utföra find, följ mängdnamns-pekarna från startnoden till en självrefererande nod träffas på!



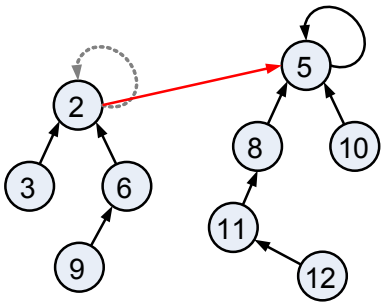


19.24

En heuristik

- Union via storlek:
  - När union utförs, låt roten i det mindre trädet peka på roten i det större

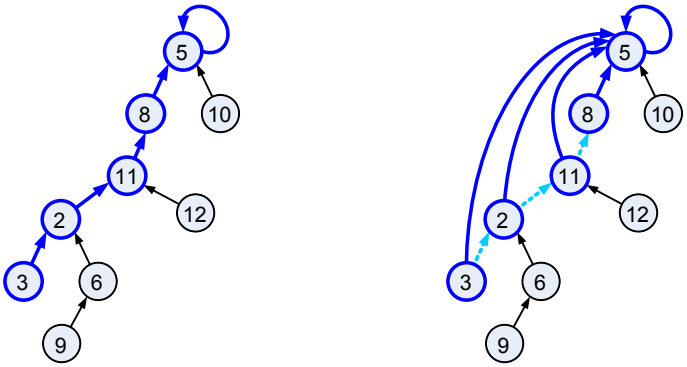
- Medför $O(n \log n)$ tid för att utföra n union- och find-operationer:
  - Varje gång vi följer en pekare kommer vi till ett delträd som är åtminstone dubbelt så stort som det förra delträdet
  - Alltså kommer vi att som mest följa $O(\log n)$ pekare för någon find



19.25

En till heuristik

- Stigkomprimering:
  - Efter att find utförts komprimera alla pekare på stigen som precis traverserats så att de alla pekar på roten



- Medför $O(n \log^* n)$ tid för att utföra n union- och find-operationer.

19.26

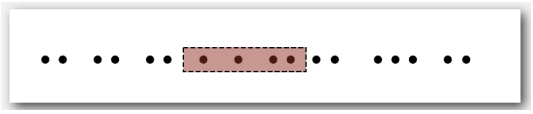
4 Geometrisk sökning

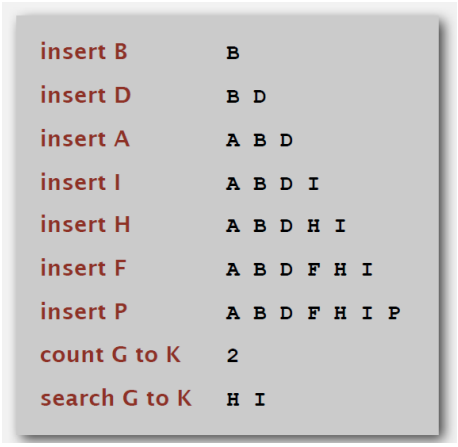
4.1 Intervallsökning

Intervallsökning i en dimension

- Utökning av ordnad symboltabell
  - Sätt in nyckel-värdepar

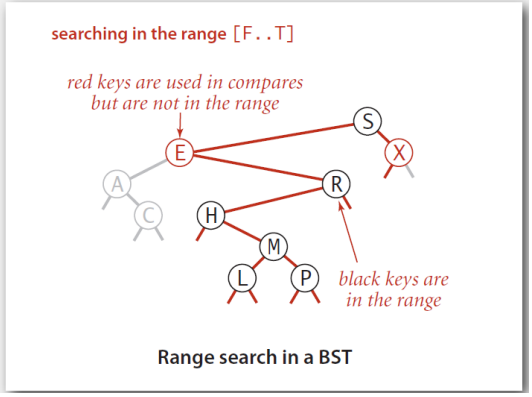
  - Sök efter nyckel k
  - Intervallsökning: hitta alla nycklar mellan k_1 och k_2
  - Intervallstorlek: antalet nycklar mellan k_1 och k_2
- Tillämpning:
  - Databasfrågor
- Geometrisk tolkning:
  - Nycklarna är punkter på en linje
  - Hitta/räkna punkter på ett givet intervall





Intervallsökning i en dimension med BST

- Hitta alla nycklar mellan k_1 och k_2
  - Hitta rekursivt alla nycklar i vänster delträd (om några kan finnas i intervallet)
  - Kontrollera nyckeln i nuvarande nod
  - Hitta rekursivt alla nycklar i höger delträd (om några kan finnas i intervallet)

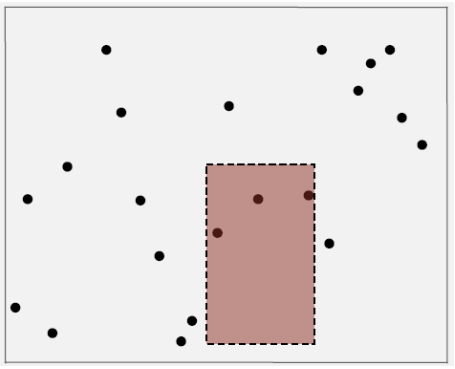


Körtiden blir proportionell mot $R + \log N$ (om vårt BST är balanserat)

Intervallsökning i två dimensioner

- Utökning av ordnad symboltabell till 2d-nycklar
  - Sätt in en 2d-nyckel
  - Sök efter en 2d-nyckel
  - Intervallsökning: hitta alla nycklar i ett 2d-intervall
  - Intervallstorlek: antalet nycklar i ett 2d-intervall
- Tillämpningar:
  - Nätverk, kretsdesign, databaser
- Geometrisk tolkning:
  - Nycklarna är punkter i planet

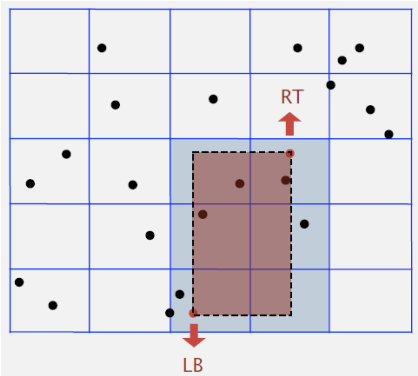
  - Hitta/räkna punkter i en given rektangel



19.29

Intervallsökning i två dimensioner med rutnät

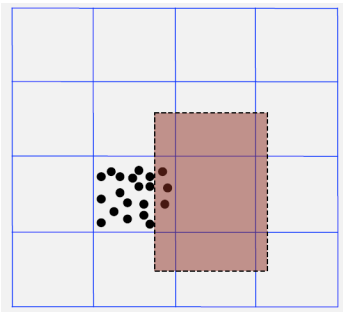
- Dela upp planet i $M \times M$-rutnät av kvadrater
- Skapa lista av punker i varje kvadrat
- Använd 2d-array för att direkt indexera relevant kvadrat
- Intervallsökning: undersök bara de kvadrater som överlappar frågan



19.30

Klustring

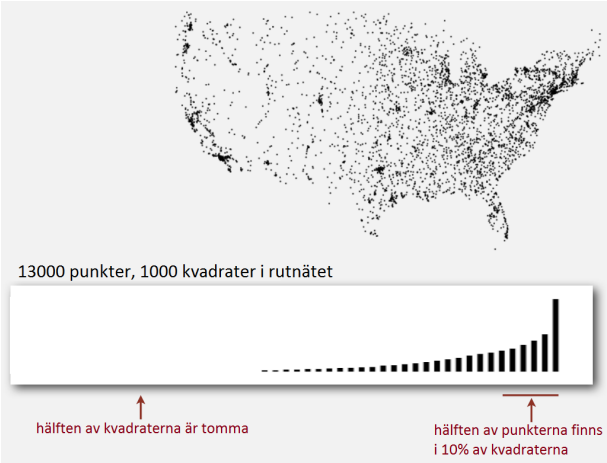
- Rutnätsimplementation:
  - Snabb, enkel lösning för väldistribuerade punktmängder
- Problem: Klustring ett välkänt problematiskt fenomen i geometriskt data
  - Listorna är för långa, trots att medellängden är kort
  - Behöver datastruktur som *anpassar* sig till data



19.31

Klustring

- Rutnätsimplementation:
  - Snabb, enkel lösning för väldistribuerade punktmängder
- Problem: Klustring ett välkänt problematiskt fenomen i geometriskt data
  - Exempel: kartdata



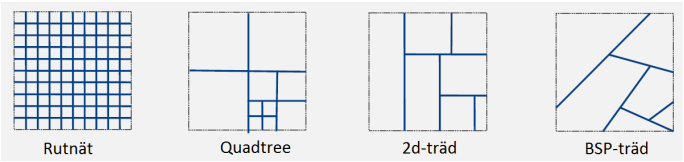
19.32

4.2 Trädstrukturer

Trädstrukturer

Använd ett *träd* för att rekursivt dela upp planet

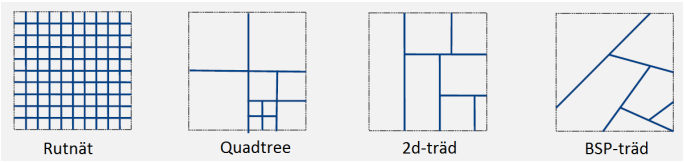
- Rutnät: Dela upp planet likformigt i kvadrater
- Quadtree: Dela upp planet rekursivt i fyra kvadranter
- 2d-träd: Dela upp planet rekursivt i två halvplan
- BSP-träd: Dela upp planet rekursivt i två regioner



19.33

Tillämpningar

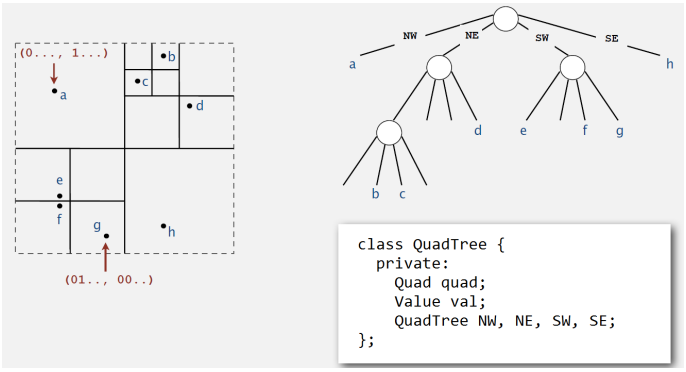
- Ray-tracing
- Intervallsökning i två dimensioner
- Flygsimulatorer
- N-kroppssimulering
- Kollisionsdetektering
- Astronomiska databaser
- Sökning efter närmaste grannar
- Adaptiv rutnätsgenerering
- Accelerera rendreringen i Doom
- Ta bort dolda ytor och skuggning



19.34

Quadtree

- Idé: Dela upp planet rekursivt i 4 kvadranter
- Implementation: 4-vägsträd (egentligen ett trie)

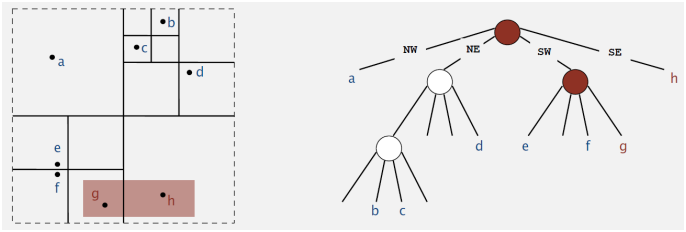


- Fördel: Bra prestanda vid klustring av data
- Nackdel: Godtyckligt djup!

19.35

Quadtree: Intervallsökning i två dimensioner

- Hitta rekursivt alla nycklar i NE-kvadranten (om några kan finnas i intervallet)
- Hitta rekursivt alla nycklar i NW-kvadranten (om några kan finnas i intervallet)
- Hitta rekursivt alla nycklar i SE-kvadranten (om några kan finnas i intervallet)
- Hitta rekursivt alla nycklar i SW-kvadranten (om några kan finnas i intervallet)

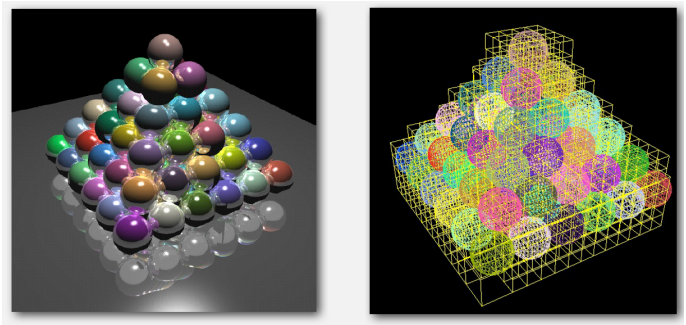


- Typisk exekveringstid: $R + \log N$

19.36

Dimensionalitetsproblemet

- Intervallsökning i k dimensioner
  - Huvudsaklig tillämpning: Multidimensionella databaser
  - 3d: Octree: dela rekursivt upp 3d-rymden i 8 oktanter
  - 100d: Centree: dela rekursivt upp 100d-rymden i 2^{100} centranter???

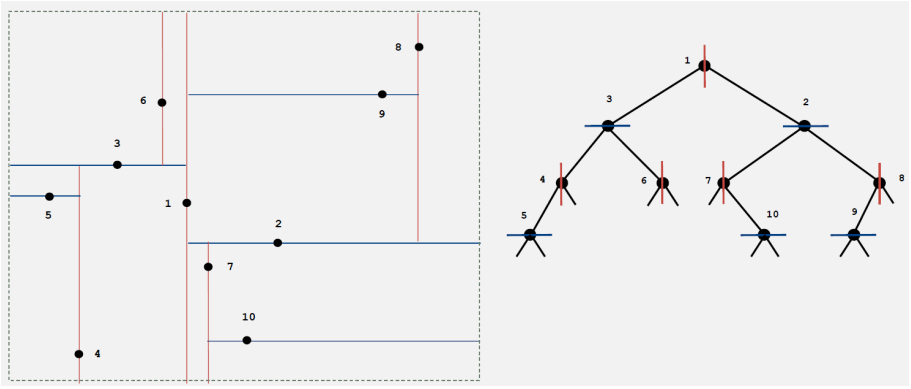


Raytracing with octrees
http://graphics.cs.ucdavis.edu/~gregorsk/graphics/275.html

19.37

2d-träd

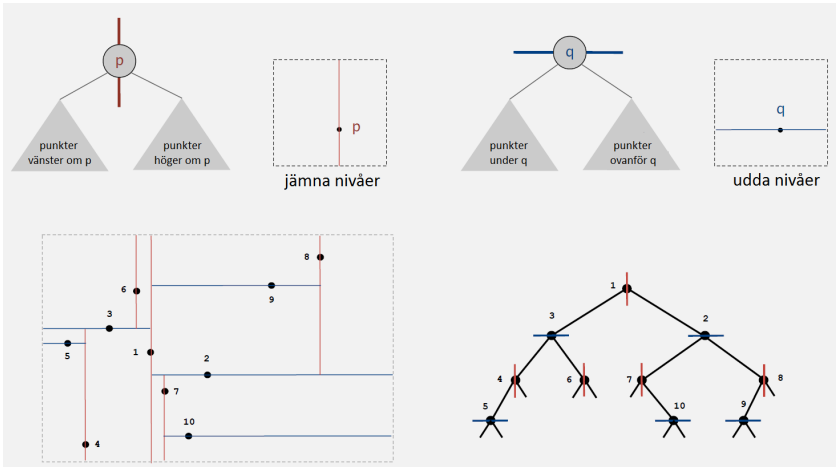
Dela rekursivt upp planet i två halvplan



19.38

2d-träd

- Datastruktur: BST, men alternera att använda x- och y-koordinat som nyckel
  - Sökning ger rektangel innehållande punkt
  - Insättning underdelar planet ytterligare

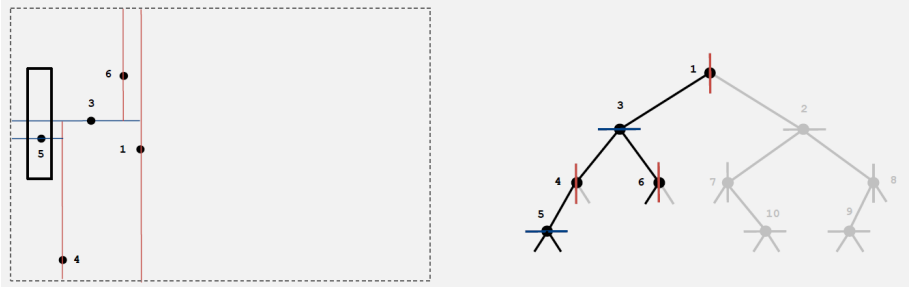


19.39

2d-träd: Intervallsökning i två dimensioner

Hitta alla punkter i frågans rektangel (i linje med koordinataxlarna)

- Kontrollera om punkt i nod ligger i given rektangel
- Sök rekursivt i vänstra/övre underdelningen (om några punkter kan finnas i rektangeln)
- Sök rekursivt i högra/undre underdelningen (om några punkter kan finnas i rektangeln)



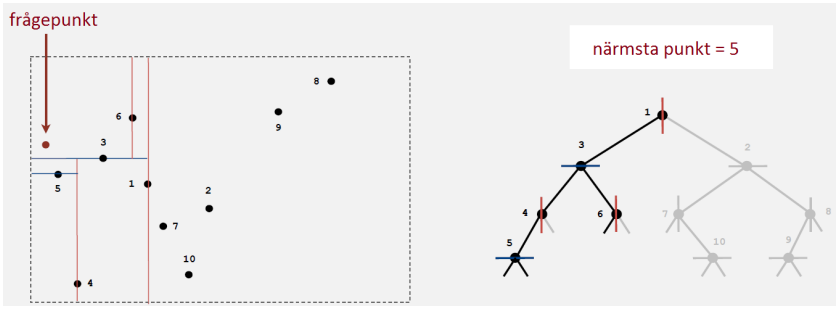
- Typisk exekveringstid: $R + \log N$
- Värsta fallet (under antagandet att trädet är balanserat): $R + \sqrt{N}$

19.40

2d-träd: Sökning efter närmsta granne

Hitta punkten närmast en given punkt

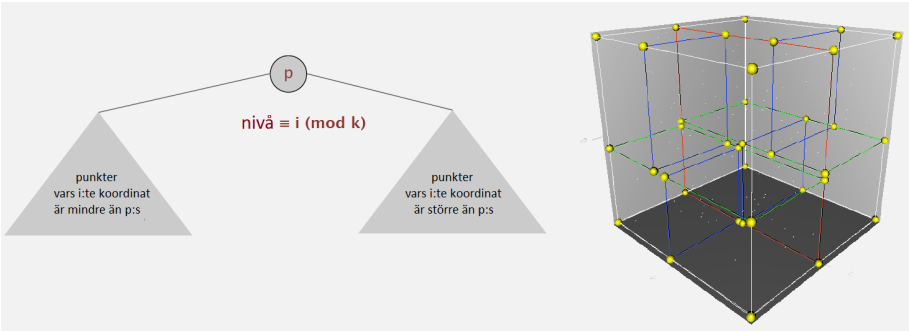
- Kontrollera avståndet från punkt i nod till frågepunkten
- Sök rekursivt i vänstra/övre underdelningen (om kan innehålla närmre punkt)
- Sök rekursivt i högra/undre underdelningen (om kan innehålla närmre punkt)
- Organisera rekursiv metod så att den börjar med att söka efter frågepunkten



- Typisk exekveringstid: $\log N$
- Värsta fallet (även om trädet är balanserat): N

Kd-träd

- Kd-träd: Partitionera rekursivt k-dimensionell rymd i två halvrymder
  - Implementation: BST, men cykla dimensionerna likt 2d-träd



- Effektiv, enkel datastruktur för att behandla k-dimensionellt data
  - Bred användning
  - Anpassar sig väl till högredimensionell och klustrig data
  - Upptäcktes av en student (Jon Bentley) i en algoritmkurs!