



Programmering för Språkteknologer II

Markus Saers
markus.saers@lingfil.uu.se
Rum 9-2040
stp.lingfil.uu.se/~markuss/ht09/pst2



Innehåll

- Associativa datastrukturer
 - Hashtabeller
 - Sökträd
- Implementationsdetaljer för att använda associativa datastrukturer



Associativa datastrukturer

- Låter oss associera godtyckliga nycklar med godtyckliga värden
 - Vektor: associerar heltal med godtyckliga värden
- Behöver inte vara "tät"
 - Vektorer: måste ha plats för alla nycklar från 0 till den högsta
- Nycklar måste vara unika
 - En nyckel får inte associeras med flera värden
 - Utgör en "mängd"
 - Vektorer: mängden giltiga index



Associativa datastrukturer

- Två vanliga implementationer
 - Hashtabeller
 - (Jätte) snabba
 - $O(1)$
 - Osorterade
 - Nästan täta
 - Sökträd
 - Snabba
 - $O(\log n)$
 - Sorterade
 - Helt täta

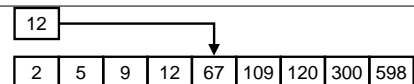


Binär sökning

- Antag att vi har en vektor med sorterade element
 - [2 5 9 12 67 109 120 300 598]
- Vi vill finna ett givet element i vektorn
 - 12
- Titta på det mittersta elementet
 - Avgör om det sökta elementet är exakt lika, större eller mindre
 - $12 < 67$
 - Avgör om vi är färdiga (lika) eller i vilken halva elementet finns ($<$ vänster, $>$ höger)



Binär sökning



UPPSALA UNIVERSITET

Binär sökning

12 != 67

UPPSALA UNIVERSITET

Binär sökning

12 != 67
12 < 67

UPPSALA UNIVERSITET

Binär sökning

UPPSALA UNIVERSITET

Binär sökning

12 != 9

UPPSALA UNIVERSITET

Binär sökning

12 != 9
12 > 9

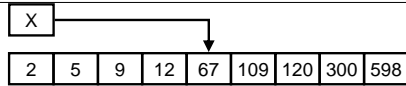
UPPSALA UNIVERSITET

Binär sökning

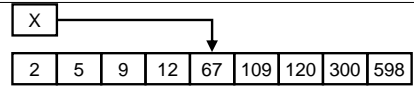
12 == 12



Binär sökning



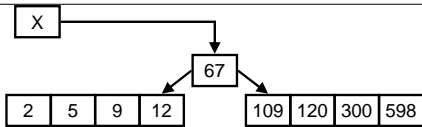
Binär sökning



Jämför X med 67
 Lika färdig
 Mindre titta till vänster
 Större titta till höger



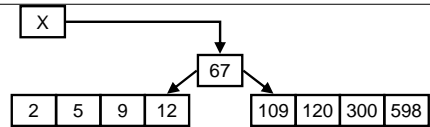
Binär sökning



Jämför X med 67
 Lika färdig
 Mindre titta till vänster
 Större titta till höger



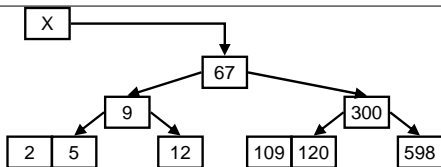
Binär sökning



Jämför X med 9 / 300
 Lika färdig
 Mindre titta till vänster
 Större titta till höger



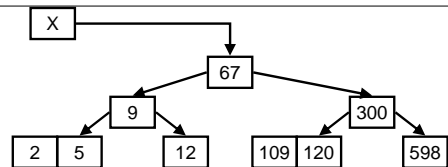
Binär sökning



Jämför X med 9 / 300
 Lika färdig
 Mindre titta till vänster
 Större titta till höger

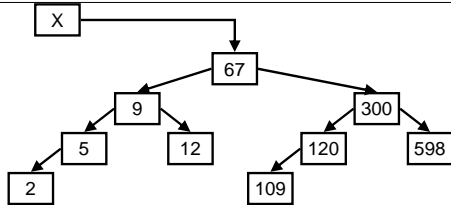


Binär sökning



Jämför X med 5 / 12 / 120 / 598
 Lika färdig
 Mindre titta till vänster
 Större titta till höger

Binär sökning: beslutsträd



Binär sökning: beslutsträd

- Vid varje nod:
 - Har vi kommit till det tal vi sökte?
 - Om ja: vi är klara!
 - Om nej: är det sökta talet större eller mindre?
 - Om större: leta i noden till höger
 - Om mindre: leta i noden till vänster
- Om vi ska vidare från en nod finns två val
 - Binär sökning
 - Beslutsträd = binärt sökträd

Sökträd

- I praktiken binärt sökträd
- Konkret modellering av beslutsträdet från en binär sökning
 - Binär sökning: algoritmen applicerad på en vektor
 - Binära sökträd: datastruktur som direkt modellerar binär sökning

Binärt sökträd

- Består av noder
- Varje nod har
 - En nyckel
 - Ett värde
 - Ett vänsterbarn som är rot i trädet med alla noder vars nycklar är mindre än nyckeln
 - Ett högerbarn som är rot i trädet med alla noder vars nycklar är större än nyckeln

Binära sökträd

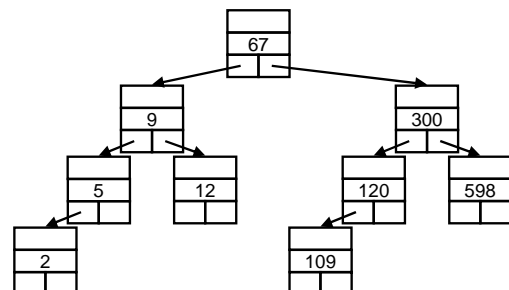
värde

 nyckel

 v. barn

 h. barn

Binära sökträd



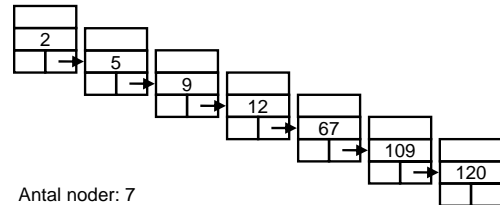


Egenskaper

- Hur många jämförelser behöver vi göra för att hitta en nyckel?
 - Höjden på trädet
 - Ett balanserat träd är $\log n$ högt: $O(\log n)$
 - $\log(9) = 3,17 = 4$



Obalanserat binärt sökträd



Antal noder: 7
 Höjd: 7
 Söktid: $O(n)$



Balanserade binära sökträd

- Varje gång en nod läggs in kontrollerar trädet att det är i "tillräcklig balans"
- Olika metoder
 - Röd-svarta träd
 - AVL-träd
- Påverkar inte tidskomplexiteten!
 - Söker på nervägen, balanserar på tillbakavägen
 - Däremot den faktiska tiden med en faktor 2



Balanserade binära sökträd

- Sökning: $O(\log n)$
- Insättning: $O(\log n)$
- Borttagning: $O(\log n)$
- Traversering: $O(n)$
 - Genomlöpning som besöker varje nod exakt en gång



Ny sorteringsalgoritm: TreeSort

- Lägg in alla värdena som nycklar i ett balanserat binärt sökträd
 - Antal nycklar: n
 - Tid per insättning: $O(\log n)$
 - Total tid: $O(n \log n)$
- I praktiken långsammare än MergeSort



Objekt som nycklar i TreeXXX

- Trädklasserna behöver något sätt att avgöra vilken ordning nycklarna kommer i
 - Naturlig ordning
 - Jämförelseklass



Naturlig ordning

- Alla klasser som implementerar `Comparable<T>` på sig själva har en naturlig ordning
- Innebär att metoden `int compareTo(T t)` implementeras
 - Ger 0 om objekten är lika
 - Ger < 0 om objektet är mindre än t
 - Ger > 0 om objektet är större än t
 - För heltal: `return value - t.value;`
- "Naturlig" = given av klassen



Exempel: Point

```
public class Point
implements Comparable<Point>{
    private int x;
    private int y;
    public int compareTo(Point p) {
        if (x == p.x) {
            return y - p.y;
        } else {
            return x - p.x;
        }
    }
}
```



Exempel: Person

```
public class Person
implements Comparable<Person> {
    private String firstName;
    private String lastName;
    public int compareTo(Person p) {
        if (firstName.equals(p.firstName)) {
            return lastName.compareTo(p.lastName);
        } else {
            return firstName.compareTo(p.firstName);
        }
    }
}
```



Olika ordningar

- Nu bestämde vi att personer ordnas efter hur deras namn läsas ut
- Vad göra om vi vill bygga en telefonkatalog?
 - Ordnad i första hand på efternamn
- Jämförelseobjekt!
 - Implementerar `Comparator<T>`



Comparator<T>

- Finns i `java.util` (måste importeras)
- Måste implementera metoden `int compare(T a, T b)`
- Ska vara som `a.compareTo(b)`
- Fast enligt den ordning som klassen representerar



Exempel: Person

```
public class Person implements Comparable<Person> {
    private String firstName;
    private String lastName;
    public int compareTo(Person p) {
        if (firstName.equals(p.firstName)) {
            return lastName.compareTo(p.lastName);
        } else {
            return firstName.compareTo(p.firstName);
        }
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
}
```



Personjämförare i telefonkatalogordning

```
import java.util.*;

public class PhoneBookPersonComparator
implements Coparator<Person> {
    public int compare(Person a, Person b) {
        if (a.getLastName().equals(b.getLastName()) {
            return
                a.getFirstName().compareTo(b.getFirstName());
        } else {
            return
                a.getLastName().compareTo(b.getLastName());
        }
    }
}
```



Användning

```
public class Test {
    public static void main(String[] args) {
        TreeSet<Person> people = new TreeSet<Person>();
        TreeMap<Person, Integer> phoneBook =
            new TreeMap<Person, Integer>(
                new PhoneBookPersonComparator()
            );
        // Lägg in personer i people och phoneBook
        for (Person p : people) {
            // Ordnade efter förnamn sedan efternamn
        }
        for (Person p : phoneBook.keySet()) {
            // Ordnade efter efternamn sedan förnamn
        }
    }
}
```



Frågor?