Föreläsning 3
Datastrukturer (DAT037)

Fredrik Lindblad[1]

6 november 2017

[1]Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se
`http://www.cse.chalmers.se/edu/year/2015/course/DAT037`

Innehåll

- Listor, Stackar, Köer
- Länkade listor
- Jämförelse av listimplementationer
- Cirkulära arrayer
- Invarianter, assertions, testning
- Träd

Listor, stackar och köer: ADT-er

ADT	Operationer
Lista	add(x), add(x, i), remove(i), get(i), set(i, x)
Stack	push(x), pop()
Kö	enqueue(x), dequeue()
Deque	addFirst(x), removeFirst(), addLast(x), removeLast()

(Endast de centrala operationerna.)

Listor, stackar och köer: datastrukturer

ADT	Implementationer
Lista	dynamisk array, enkellänkad lista, dubbellänkad lista
Stack	lista
Kö, Deque	länkad lista, cirkulär array

Stackar och köer: tillämpningar

Stackar

- Implementera rekursion.
- Evaluera postfix-uttryck.
- ...

Köer

- Skrivarköer.
- Bredden först-sökning (grafalgoritm).
- ...

Länkade listor

Länkade listor

Många varianter:

- Objekt med pekare till första noden, kanske storlek.
- Pekare till sista noden?
- Enkellänkad, dubbellänkad?
- Vaktposter (sentinels)? Först/sist/både och?

Länkade listor

Många varianter:

- Objekt med pekare till första noden, kanske storlek.
- Pekare till sista noden?
- Enkellänkad, dubbellänkad?
- Vaktposter (sentinels)? Först/sist/både och?

Vi tar dubbellänkad lista med dubbla vaktposter som exempel. Det gäller att vara noggrann när man utför den s.k. pekarjongleringen.

Dubbellänkad lista med dubbla vaktposter

```
public class DoublyLinkedList<A> {
    private class ListNode {
        A contents;
        ListNode next;  // null omm sista vakten.
        ListNode prev;  // null omm första vakten.
    }

    ...
}
```

Dubbellänkad lista med dubbla vaktposter

```
public class DoublyLinkedList<A> {
    ...

    private ListNode first, last;  // Ej null.
    private int size;

    // Konstruerar tom lista.
    public DoublyLinkedList() {
        first      = new ListNode();
        last       = new ListNode();
        first.next = last;
        last.prev  = first;
        size       = 0;
    }
}
```

Implementera metoden `add` som lägger till ett element i slutet av listan.

```
void add(A x) {
  ...
}
```

```
void add(A x) {
 ListNode l = new ListNode();
 l.contents = x;
 l.next = last;
 l.prev = last.prev;
 last.prev.next = l;
 last.prev = l;
 size++;
}
```

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`		
`add(x, i)`		
`remove(x)`		
`remove(i)`		
`get(i)`		
`set(i, x)`		
`contains(x)`		
`size`		
`iterator`		
`hasNext/next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`		
`remove(x)`		
`remove(i)`		
`get(i)`		
`set(i, x)`		
`contains(x)`		
`size`		
`iterator`		
`hasNext`/`next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`		
`remove(i)`		
`get(i)`		
`set(i, x)`		
`contains(x)`		
`size`		
`iterator`		
`hasNext/next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$
`remove(i)`		
`get(i)`		
`set(i, x)`		
`contains(x)`		
`size`		
`iterator`		
`hasNext`/`next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$
`get(i)`		
`set(i, x)`		
`contains(x)`		
`size`		
`iterator`		
`hasNext/next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$
`set(i, x)`		
`contains(x)`		
`size`		
`iterator`		
`hasNext/next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$
`contains(x)`		
`size`		
`iterator`		
`hasNext/next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$
`size`		
`iterator`		
`hasNext/next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$
`iterator`		
`hasNext/next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$
`iterator`	$O(1)$	$O(1)$
`hasNext/next`		
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$
`iterator`	$O(1)$	$O(1)$
`hasNext/next`	$O(1)$	$O(1)$
`remove`		

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista
`add(x)`	$O(1)$ am	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$
`iterator`	$O(1)$	$O(1)$
`hasNext/next`	$O(1)$	$O(1)$
`remove`	$O(n)$	$O(1)$

Cirkulära arrayer

Cirkulära arrayer

Alternativ till länkade listor för implementering av köer.

Cirkulära arrayer

Alternativ till länkade listor för implementering av köer.

- Utgår från en dynamisk array.
- Två indexpekare; en för början av kön och en för slutet.
- Behöver hålla reda på om kön är tom eller full när båda pekar på samma element.
- När man kopierar till ett större array får man ta hänsyn till vilka element som används i den gamla.

Cirkulära arrayer

```
public static class CircularArrayQueue<A> {
    private A[] queue;
    private int size;
    private int front;  // nästa element
    private int rear;  // nästa lediga

    public CircularArrayQueue(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException(
             "non-positive capacity");
        }
        queue = (A[]) new Object[capacity];
        size = front = rear = 0;
    }
    ...
```

Cirkulära arrayer

```
public void enqueue(A a) {
    if (size == queue.length) {
        doubleCapacity();
    }

    size++;
    queue[rear] = a;
    rear = (rear + 1) % queue.length;
}
```

Cirkulära arrayer

```
public A dequeue() {
    if (size == 0) {
        throw new NoSuchElementException(
         "queue empty");
    }

    size--;
    A a = queue[front];
    queue[front] = null;  // undvik minnesläckor
    front = (front + 1) % queue.length;

    return a;
}
```

Cirkulära arrayer

```
    private void doubleCapacity() {
        A[] newQueue =
            (A[]) new Object[2 * queue.length];

        for (int i = 0, j = front; i < size;
                i++, j = (j + 1) % queue.length) {
            newQueue[i] = queue[j];
        }
        queue = newQueue;
        front = 0;
        rear  = size;
    }
}
```

Testning

Invariant

- Egenskap som ”alltid” gäller för programmets tillstånd.
- Exempel:
  1. `first != null`.
  2. `last.next = null`.
  3. `n.next.prev = n` (om `n` ej är vaktpost).
  4. $\texttt{first.next}^{\texttt{size}+1} = \texttt{last}$.
- Invarianter bryts ibland temporärt när objekt uppdateras.

Precondition

- Krav som förväntas vara uppfyllt
  då metod anropas.
- Exempel: `pop` kräver att stacken inte är tom.

Postcondition

- Egenskap som är uppfylld efter anrop, givet preconditions och invarianter.
- Exempel: Efter `push` är stacken inte tom.

Assertion

Man kan testa vissa egenskaper med ”assertions”.
Kan vara smidigt för att hitta/undvika fel i labbar.

Fail.java

```
public class Fail {
  public static void main (String[] args) {
    assert args.length == 2;
  }
}
```

```
$ javac Fail.java
$ java Fail
$ java -ea Fail ett två
$ java -ea Fail
Exception in thread "main" java.lang.AssertionError
        at Fail.main(Fail.java:3)
```

Assertion

```
// Skapar ny /intern/ listnod.
// Precondition: prev != null && next != null.
ListNode(A contents, ListNode prev, ListNode next) {
  assert prev != null && next != null;

  this.contents = contents;
  this.prev     = prev;
  this.next     = next;
}
```

Nu leder

```
new ListNode(x,first.prev,first.next)
```

till ett `AssertionError` (med `-ea`),
inte ett kryptiskt fel senare.

Assertion

```
// Lägger till x efter n.
// Precondition: n != null && n != last.
void addAfter(ListNode n, A x) {
    assert n != null && n != last;

    ListNode next = n.next;
    n.next        = new ListNode(x, n, next);
    next.prev     = n.next;
    size++;
}

void addFirst(A x) {
    addAfter(first, x);
}
```

Exempel på metod som kontrollerar invarianter hos en instans av den dubbellänkade listan vi tittade på tidigare.

```
private void checkInvariants() {
 assert first != null && last != null;
 assert first.prev == null;
 assert last.next == null;
 assert size >= 0;
 ListNode l = first;
 for (int i = -1; i < size; i++) {
  assert l.next != null;
  assert l.next.prev == l;
  l = l.next;
 }
 assert l == last;
}
```

Korrekthet

Hur kan man förvissa sig om att man löst en uppgift korrekt?

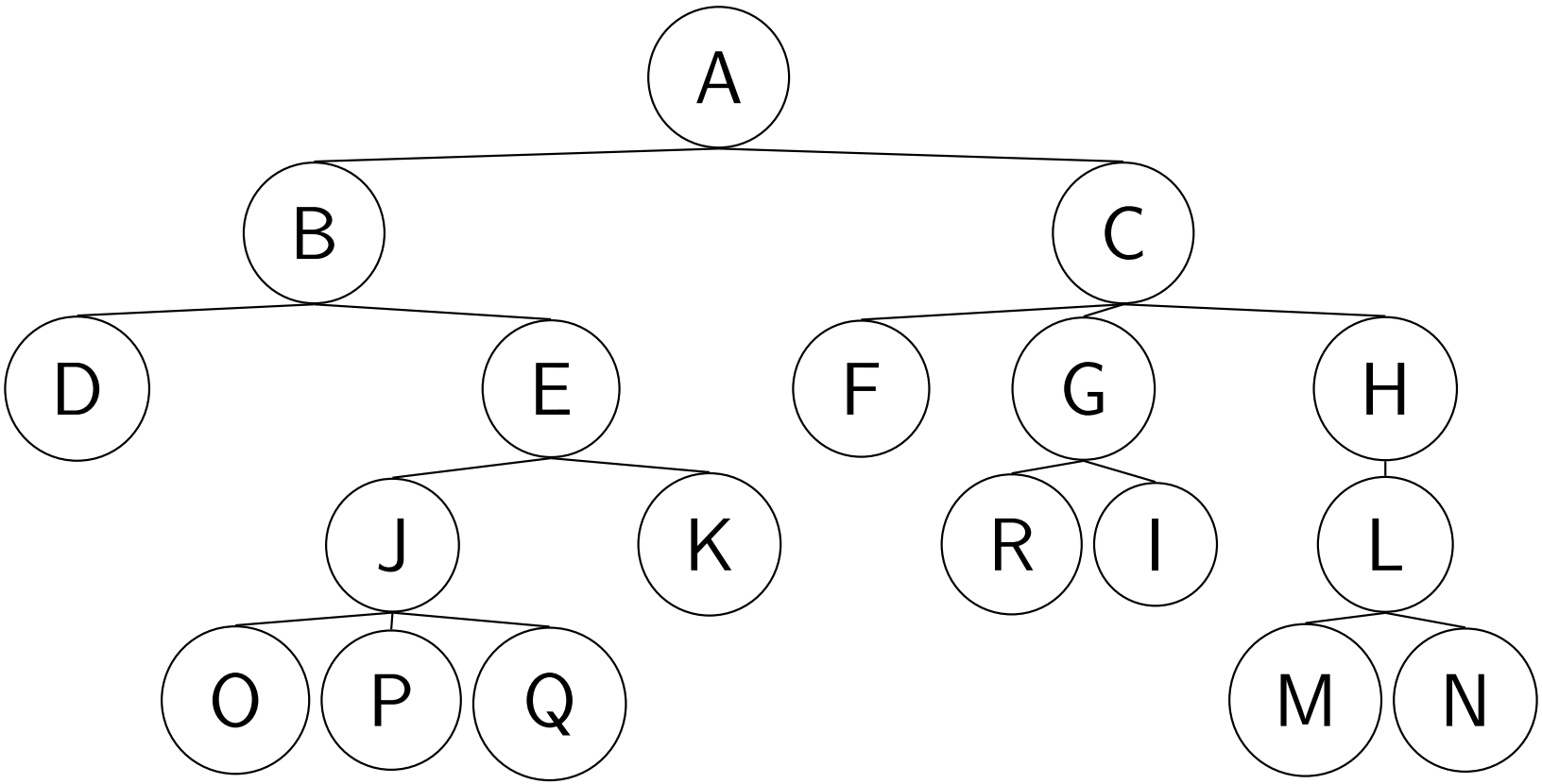
- Bevis. Svårt, ta mycket tid.
- Tester. Enklare och snabbare, men ger ingen garanti för korrekt. Snarare ett sätt att upptäcka fel.

Träd

Träd

Är ytterligare ett exempel på en ADT. Är dock inte så tydligt vilken standarduppsättningen av metoder skulle vara. Därför är träd är inte en ADT (ett interface) i java collection framework. Olika former av träd förekommer däremot som representation i implementeringar för andra ADTer.

Abstrakt beskrivning



Termer

- Träd(trees) består av *noder*, som har 0 eller flera underordnade noder/delträd. Noderna är förbundna med *bågar*(edges).
- Noden som är underordnad en annan kallas *barn*(child), den överordnade noden för *förälder*(parent). En indirekt överordnad nod kallas *förfader*(ancestor) och indirekt underordnad för *avkomling*(descendant). Noder som har samma förälder kallas *syskon*(siblings).
- I varje träd finns en nod som är *rot*. Den har ingen förälder. Alla andra noder i ett träd har en förälder. Noder som inte har några barn kallas *löv*(leaves).

- Trädstrukturer brukar ritas upp-och-ner jämför med riktiga träd, d.v.s med roten högst upp.
- En mängd av träd kallas *skog*(forest).
- I ett *ordnat* träd har syskonen en inbördes ordning. Motsatsen är *oordnat* träd. Jmf. lista gentemot mängd.

Användningsområden

Filsystem, klassificering med kategorier och underkategorier, representation för matematiska uttryck, programmeringsspråk och naturliga språk.

Användningsområden

Filsystem, klassificering med kategorier och underkategorier, representation för matematiska uttryck, programmeringsspråk och naturliga språk.

Som representation för ADTer används det till sorterade samlingar och avbildningar samt prioritetsköer, t.ex. `PriorityQueue` och `TreeMap` i java.

Fler definitioner

Längden(length) av en *väg*(path) mellan två noder = antal passerade bågar.

Djupet(depth) av en nod är längden till roten.

Höjden(height) av en nod är längsta vägen ner till ett löv. Höjden av ett träd = rotens höjd.

Storleken(size) av ett träd är antalet noder.

Implementering

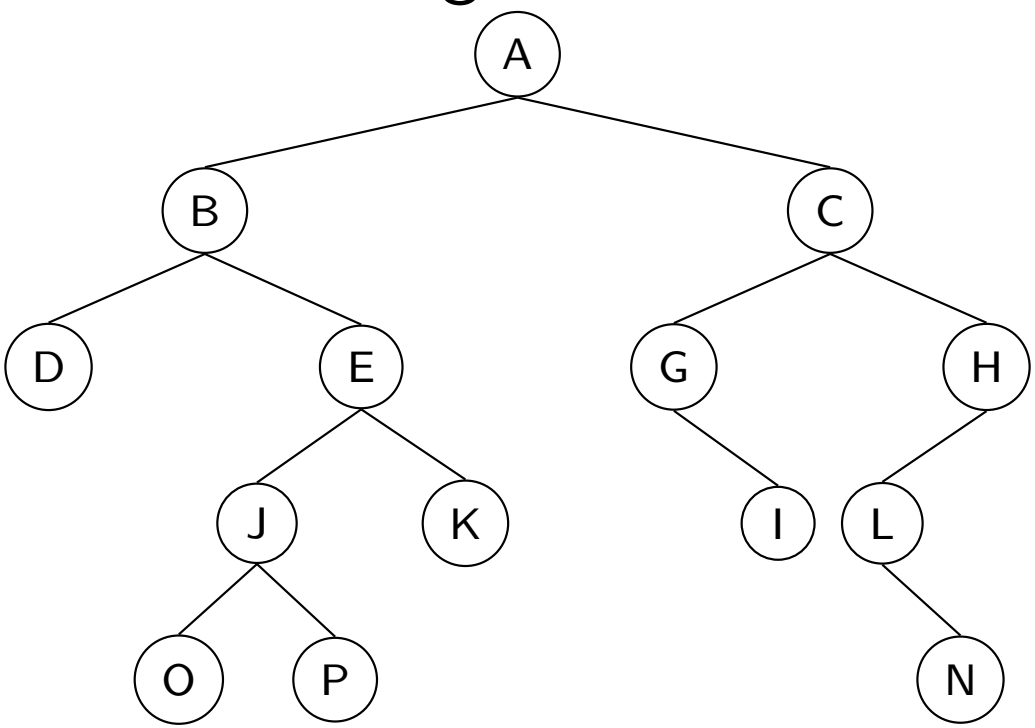
- Pekare – motsvarar länkade listor för listor. Detta funkar för alla typer av träd.
- Array – effektivt för vissa typer av träd, t.ex. binära heapar som vi kommer titta på nästa gång.

Representation med pekare

- En lokal klass för noder.
- I huvudklassen en pekare till rotnoden.
- I nodklassen en pekare till varje barn.
- Ibland pekare till föräldern.
- Om obegränsat antal barn – lista (ordnad), mängd (oordnad), eller i varje nod en pekare till vänstra barnet och syskonet till höger.

Binära träd

Binära träd är en vanlig typ av träd. Varje nod har max två barn och man skiljer på en nod med bara ett delträd till vänster och samma nod med samma delträd till höger.



Sorterade och balanserade träd

Många typer av träd är sorterade, d.v.s. elementen i noderna uppfyller någon ordning (t. ex. enligt en `Comparator`), och balanserade, d.v.s. uppfyller vissa krav på hur höjden av delträd förhåller sig till varandra. Vi kommer se ett exempel på detta nästa gång och fler när vi behandlar sökträd om ett par veckor.

Trädgenomlöpning (Tree traversal)

Besöker alla noder i ett träd.

- Pre-order – först noden själv, sedan vänster delträd och sist höger delträd
- In-order – vänster, noden, höger
- Post-order – vänster, höger, noden
- Bredden först – Först roten, sedan nodenerna på nästa nivå, etc.

Antag binärt träd med följande representation.

```
class Tree<E> {
 private class Node {
  E data;
  Node left = null, right = null;
 }

 Node root = null;
 ...
 public void traversePrint() {
  ...
 }
}
```

Skriv en metod som genomlöper alla noder pre/in/post-order och skriver ut datan.

```
public void traversePrint() {
 printSubtree(root);
}

private void printSubtree(Node n) {
 if (n == null) return;
 System.out.println(n.data.toString());
 printSubtree(n.leftchild);
 printSubtree(n.rightchild);
}
```

Ovan ger pre-order traversal. In-order och post-order fås genom att sätta raden som skriver ut mellan resp. efter de rekursiva anropen.

Rekursion

- Rekursiv metod anropar sig själv.
- Rekursiva metoder som anropar sig själv en gång går utan större problem att skriva ut som icke-rekursiv med slinga. Men kan vara naturligare med rekursion.
- Vid flera rekursiva anrop, t.ex. vid trädtraversering, är det ofta krångligt att implementera på annat sätt.
- Tänk på att det måste finnas ett slutvillkor så anropen inte sker i oändlighet.