Inhaltsverzeichnis

1 Datenstrukturen für Sequenzen

Sequenzen

Felder

Listen

Stacks und Queues

Sortierte Sequenzen

* 1. **Sequenzen**

**1.1.1Sequenz: lineare Struktur**

s = ⟨e0, . . . , en−1⟩

(Gegensatz: verzweigte Struktur in Graphen, fehlende Struktur in Hashtabelle)

**1.1.2Klassische Repräsentationen:**

**• (Statisches) Feld / Array:** direkter Zugriff über s[i]

• Vorteil: Zugriff über Index, homogen im Speicher

• Nachteil: dynamische Größenänderung schwierig

**• Liste:** indirekter Zugriff über Nachfolger / Vorgänger

• Vorteil: Einfügen / Löschen von Teilsequenzen

• Nachteil: kein Zugriff per Index, Elemente über Speicher verteilt

**1.1.3Operationen:**

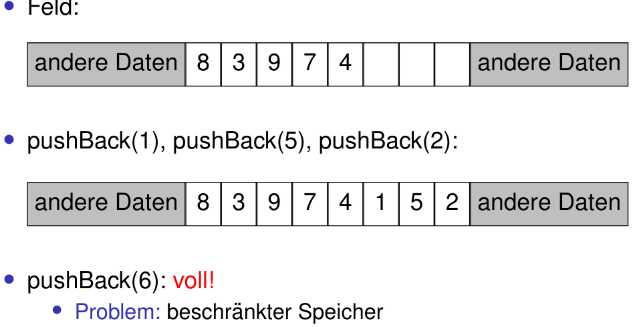
• ⟨e0, . . . , en−1⟩.get(i) = ei

• ⟨e0, . . . , ei−1, ei, . . . , en−1⟩.set(i, e) = ⟨e0, . . . , ei−1, e, . . . , en−1⟩

• ⟨e0, . . . , en−1⟩.pushBack(e) = ⟨e0, . . . , en−1, e⟩

• ⟨e0, . . . , en−1⟩.popBack() = ⟨e0, . . . , en−2⟩

• ⟨e0, . . . , en−1⟩.size() = n

**1.2 Felder(Kettung, Array)**

Problem:

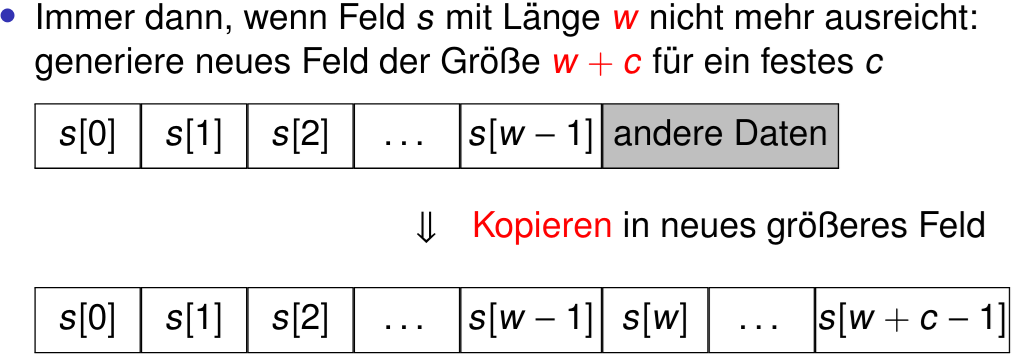
• Beim Anlegen des Felds ist nicht bekannt, wie viele Elemente es enthalten wird

• Nur Anlegen von statischen Feldern mit fester Länge arr.length möglich

**(s = new ElementTyp[arr.length])**

Lösung: Datenstruktur für dynamisches Feld

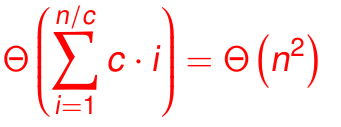
**1.2.1 Dynamische Feld**

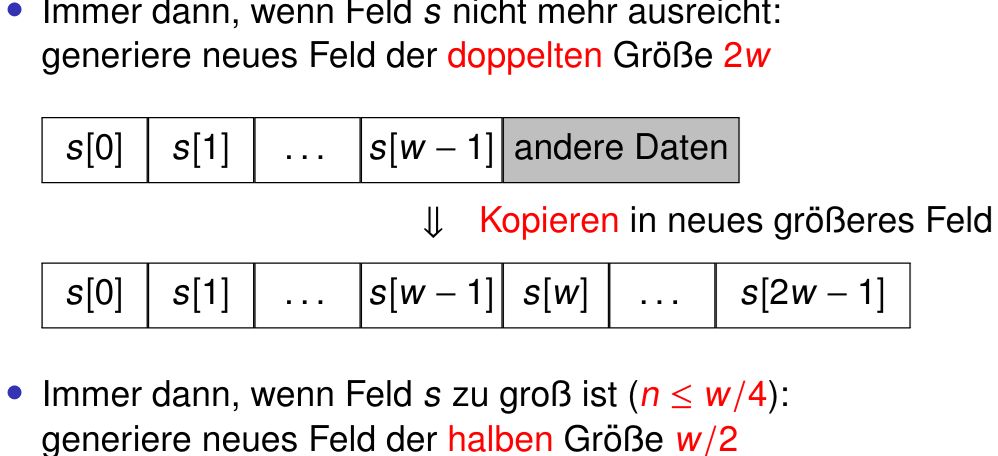
1.2.1.1Erste Idee:

Zeitaufwand für Erweiterung: Θ(w)

Zeitaufwand für n pushBack Operationen:

• Aufwand von Θ(w) nach jeweils c Operationen (wobei w immer größer wird)

• Gesamtaufwand (Start mit leerem Feld):

1.2.1.2Bessere Idee:

1.2.1.3: Implementierung:

Klasse **UArray** mit den Methoden:

• ElementTyp **get**(int i)

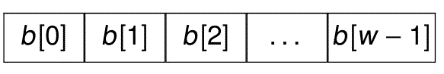
• void **set**(int i, ElementTyp e)

• int **size**()

• void **pushBack**(ElementTyp e)

• void **popBack**()

• void **reallocate**(int new w)

Klasse UArray mit den Elementen:

• β = 2 // Wachstumsfaktor

• α = 4 // max. Speicheroverhead

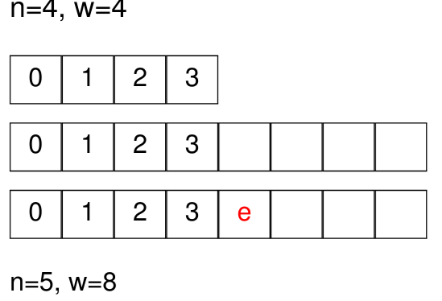
• w = 1 // momentane Feldgröße

• n = 0 // momentane Elementanzahl

• b = new ElementTyp[w] // statisches Feld

**ElementTyp get (int i) {assert(0≤i<n); return b[i];}**

**void set (int i, ElementTyp e) {assert(0≤i<n); b[i] = e;}**

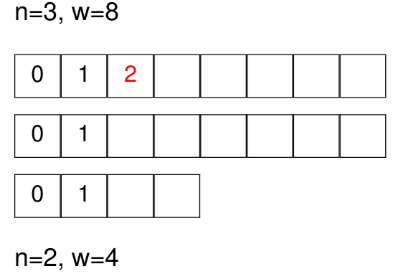
**int size () {return n;}**

**void pushback (ElementTyp e) {**

**if (n==w) {reallocate(β\*w);}**

**b[n]=e;**

**n++;}**

**void popBack () {**

**assert(n>0);**

**n−−;**

**if (α\*n ≤w && n>0) {reallocate(w/β);}**

**}**

**void reallocate (int new w) {**

**w = new w;**

**ElementTyp[ ] new b = new ElementTyp[new w];**

**for (i=0; i<n; i++) {new b[i] = b[i];}**

**b = new b;**

**}**

1.2.1.4Ü:  
Wieviel Zeit kostet eine Folge von n pushBack/popBack Operationen?

**Erste Idee: WORST CASE**

• einzelne Operation kostet O(n)最坏情况，每次都需要扩容并移动数据

• Schranke kann nicht weiter gesenkt werden, denn reallocate-Aufrufe kosten jeweils Θ(n)

⇒ also Gesamtkosten für n Operationen beschränkt durch n · O(n) = O(n^2) 需要执行n次相同操作

**Zweite Idee: NORMAL CASE**

• betrachtete Operationen sollen direkt aufeinander folgen

• zwischen Operationen mit reallocate-Aufruf gibt es immer auch welche ohne

⇒ vielleicht ergibt sich damit gar nicht die n-fache Laufzeit einer Einzeloperation并不是每次都是满数组

LEMMA:

Betrachte ein anfangs leeres dynamisches Feld s.均摊分析

Jede Folge σ = ⟨σ1, . . . , σn⟩ von pushBack und popBack Operationen auf s kann in Zeit **O(n)** bearbeitet werden. n次操作的复杂度

⇒ **Durchschnittlich** konstante Laufzeit pro Operation

• Kosten teurer Operationen werden mit Kosten billiger Operationen verrechnet.

• Man nennt das dann amortisierte Kosten bzw. amortisierte Analyse.

• In diesem Beispiel hätten wir also eine amortisierte Laufzeit von **O(1)** für die pushBack und die popBack Operation. 平均下来每次操作的复杂度都是O(1)

1.2.1.5 Formale Verrechnung: Zeugenzuordnung

• reallocate kann eine Vergrößerung oder Verkleinerung sein 重新分配大小可大可小

• reallocate als Vergrößerung von n auf 2n Speicherelemente (erfordert n Kopien):一倍扩容复制原有n个到新的去

es werden die n/2 vorangegangenen pushBack Operationen zugeordnet n个元素的复制成本被之前发生的n/2次（或大约n/2次）pushBack操作所“分摊”。这是因为在动态数组的扩容策略中，每次扩容前通常已经执行了接近当前容量一半的pushBack操作。

• reallocate als Verkleinerung von n auf n/2 Speicherelemente (erfordert n/4 Kopien):

es werden die n/4 vorangegangenen popBack Operationen zugeordnet

⇒ kein pushBack / popBack wird mehr als einmal zugeordnet 不会多次分摊成本

• Idee: verrechne reallocate-Kosten mit pushBack/popBack-Kosten

(ohne reallocate)

• Kosten für pushBack / popBack: O(1)

• Kosten für reallocate(k\*n): O(n) k：扩容倍数

• Konkret:

• Θ(n) Zeugen pro reallocate(k∗n) 每次扩容操作有Θ(n)个“见证者”

• verteile O(n)-Aufwand gleichmäßig auf die Zeugen 平均分摊

• Gesamtaufwand: **O(m**) bei m Operationen

Kontenmethode

• günstige Operationen zahlen Tokens ein

• teure Operationen entnehmen Tokens

• Tokenkonto darf **nie negativ** werden!

Bsp.:

• günstige Operationen zahlen Tokens ein

• pro pushBack 2 Tokens

• pro popBack 1 Token

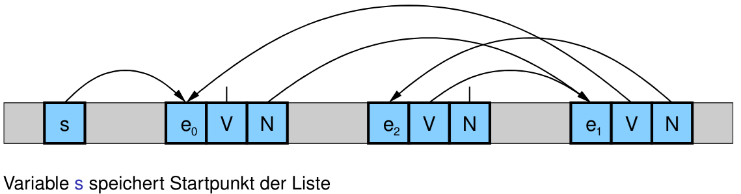
• teure Operationen entnehmen Tokens

• pro reallocate(k∗n) −n Tokens

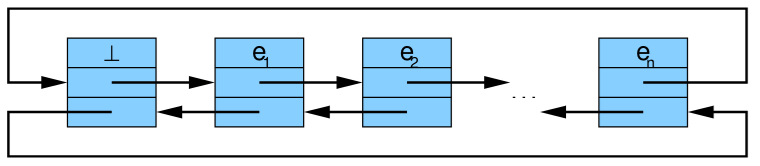
• Tokenkonto darf nie negativ werden!

• Nachweis über Zeugenargument

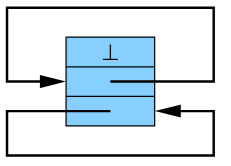
目标:在执行 push 和 pop 操作时尽可能少地触发 reallocate。这通常是通过使用动态数组的扩展策略来实现的，例如当数组满时，将其大小加倍，而不是仅增加一个元素的空间。这样，可以在多次 push 操作之后才需要进行一次 reallocate，从而减少了昂贵的内存分配和复制操作的次数。

1.3 Listen

1.3.1 Doppelt verkettete Liste



Einfache Verwaltung:

durch Virtuelle-Element h ohne Inhalt (⊥):

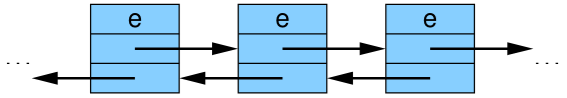
Leere Liste:

1.3.2 Implementierung:

type Handle: Item<Ele>; //声明，Item：名称， Ele：泛型

//结构体定义

type Item<Ele> {

Ele e; //Value

Handle next; //指针，指向下一个

Handle prev; //指针，指向前一个

}//实现

class List<Ele> {

Item<Ele> h; // initialisiert mit ⊥ und Zeigern auf sich selbst //定义了head为起始元素

. . . weitere Variablen und Methoden . . .

}

Invariante: //等价关系，前一个指向的next和后一个的prev都是自己

next.prev == prev.next == this

1.3.2.1 Splicing拼接：裁定特定节点序列并修改位置为特定位置

Zentrale statische Methode: splice(Handle a, Handle b, Handle t)//a:起点，b：结束节点，t：将拼接的序列粘贴到的目标节点之后。

• ⟨a, . . . , b⟩ muss Teilsequenz sein (a=b erlaubt) //序列ab必须存在于原始链表

• b nicht vor a (also Dummy h nicht zwischen a und b) //节点 b 不能位于节点 a 之前。这也意味着如果链表中使用了哑节点（Dummy Node）h（通常用作链表的头节点），那么 h 不能位于 a 和 b 之间。

• t nicht in Teilliste ⟨a, . . . , b⟩, aber evt. in anderer Liste //节点 t 不能位于子序列 ⟨a, . . . , b⟩ 中，但可以是其他链表中的节点

• splice entfernt ⟨a, . . . , b⟩ aus der Sequenz und fügt sie hinter Item t an //splice 方法将子序列 ⟨a, . . . , b⟩ 从原始链表中移除，并将其粘贴到节点 t 之后。

Bsp：

Für ⟨e1, . . . , a′, a, . . . , b, b′, . . . , t, t′, . . . , en⟩

liefert splice(a,b,t) ⟨e1, . . . , a′, b′, . . . , t, a, . . . , b, t′, . . . , en⟩

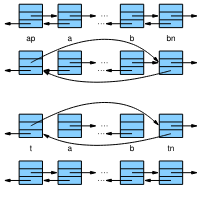
Änderung:

子序列 ⟨a, . . . , b⟩ 从原始位置被移除。

它被粘贴到了节点 t 之后。

原始链表中 a 和 b 之间的节点（如果有的话）以及 b 之后的节点（直到 t）现在位于 ⟨a, . . . , b⟩ 之后。

节点 a′ 和 b′ 分别表示在 a 和 b 之前的节点和 b 之后的节点（在调用 splice 之前）。它们现在分别位于新位置的左侧和右侧。

Implementierung: //注意双向指针

Splicing

static void splice(Handle a, b, t) {

// schneide⟨a, . . . ,b⟩ heraus

Handle ap = a.prev;

Handle bn = b.next;

ap.next = bn; //指针指向修改

bn.prev = ap; //指针指向修改

// füge ⟨a, . . . ,b⟩ hinter t ein

Handle tn = t.next; //创建指针

b.next = tn;

a.prev = t;

t.next = a;

tn.prev = b;

}

Handle head(){return h;}

boolean isEmpty(){return (h.next == head());}

Handle first(){return h.next; // evtl.h}

Handle last(){return h.prev; // evtl.h}

1.3.2.2 Verschieben\_Methoden

void moveAfter (Handle b, Handle a) {splice(b, b, a); } //b移动到a后

void moveToFront (Handle b) {moveAfter(b, head()); } //b移动到最前

void moveToBack (Handle b) {moveAfter(b, last()); } //b移动到最后

1.3.2.3 Löschen von Elementen:

mittels separater Liste freeList ⇒ bessere Laufzeit (da Speicherallokationen teuer)

void remove(Handle b){moveAfter(b, freeList.head());}

void popFront(){remove(first());}

void popBack(){remove(last());}

//Erklärung:随着对象的创建和销毁，堆内存中也会出现碎片。Java的垃圾回收器会尝试优化内存的使用，减少碎片，但并不能完全避免。

对于链表这种数据结构，如果频繁地进行节点的添加和删除操作，那么直接使用null来断开链表的链接（即“逻辑删除”）确实会导致内存碎片的问题。因为这些被“删除”的节点实际上仍然占据着内存空间，但由于它们不再是链表的一部分，因此这些内存空间变得难以被重新利用。

为了避免这种情况，可以使用“对象池”（Object Pool）或“空闲列表”（Free List）等技术来管理这些被“删除”的节点。当需要从链表中删除一个节点时，不是直接将其设置为null，而是将其放入对象池或空闲列表中。当需要添加新节点到链表时，首先检查对象池或空闲列表中是否有可用的节点。如果有，就复用这些节点，而不是重新分配新的内存空间。

1.3.2.4 Einfügen

Handle insertAfter(Elem x, Handle a) {

checkFreeList(); // u.U. Speicher allokieren //检查是否存在可回收利用的节点

Handle b = freeList.first();

moveAfter(b, a);

b.element = x;

return b;

}

//优先使用freeList中不被需要的元素，避免动态分配新的元素，降低内存调用； 修改节点指针，并在freeList中同步操作（移除该处的记录）； 修改二次利用节点的数据，符合操作规范

Handle insertBefore(Elem x, Handle b) {return insertAfter(x, b.prev);}

Handle pushFront(Elem x) { return insertAfter(x, head()); }

Handle pushBack(Elem x) { return insertAfter(x, last()); }

1.3.2.5 Suchen:

Trick: verwende Dummy-Element

Handle findNext(Elem x, Handle from) {

h.e = x;

while (from.e != x)

from = from.next;

[h.e = ⊥;]

return from;

}

//Zusatz:

关于 Dummy-Element（虚拟元素或哑节点），它是一种在链表操作中常见的技巧。在某些情况下，为了方便处理链表的边界情况（如空链表或头节点），可以在链表的开头或结尾添加一个额外的节点，这个节点通常不存储有效的数据，只用于简化操作。这个额外的节点就是 Dummy-Element。

例如，在插入一个节点到链表的开头时，如果链表为空，我们通常需要特殊处理这种情况（比如先创建一个新的头节点）。但是，如果我们使用了一个 Dummy-Element 作为链表的永久头节点，那么无论链表是否为空，我们都可以直接在这个 Dummy-Element 后面插入新的节点，从而简化了操作。同样，在删除头节点或遍历链表时，Dummy-Element 也可以简化边界条件的处理。

1.3.2.6 Erweiterung für Einfach Verkettete List

type SHandle: SItem<Elem>;

type SItem<Elem> {

Elem e;

SHandle next;

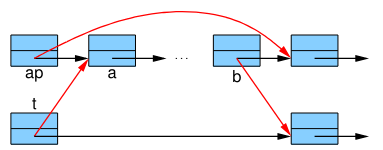
}

class SList<Elem> {

SItem<Elem> h;

. . . weitere Variablen und Methoden . . .

}

static void splice(SHandle ap, SHandle b, SHandle t) {

SHandle a = ap.next;

ap.next = b.next;

b.next = t.next;

t.next = a;

}

Wir brauchen hier den Vorgänger ap von a!

• findNext sollte evtl. auch nicht den nächsten Treffer, sondern dessen Vorgänger

liefern (damit man das gefundene SItem auch löschen kann, Suche könnte

dementsprechend erst beim Nachfolger des gegebenen SItems starten)

• Auch einige andere Methoden brauchen ein modifiziertes Interface

• Sinnvoll: Pointer zum letzten Item

⇒ pushBack in O(1)

1.4 Stacks und Queues

1.4.1 Grund Sequenzbasierte Datenstruktur

Stack(Stapel): Last In First Out

Queue: First In First Out

Deque(Double Ended Queue)

//双端队列，同时具有队列和栈特性的数据结构； Deque可以在队列的两端（头部和尾部）进行插入和删除操作。这种特性使得Deque既可以用作队列（先进先出，FIFO），也可以用作栈（后进先出，LIFO）。

1.4.1.2 Methode basierend auf Ds: Stapel:

pushBack(bzw. push) //压栈，新元素成为栈顶元素，栈的大小（或称为深度）增加1。

popBack(bzw. pop) //弹栈，栈顶元素被移除，栈的大小（或称为深度）减少1。

//如果栈已经为空，则进行pop操作可能会导致错误

last(bzw. top) //返回栈顶元素的引用（或值），但不移除该元素

Queue：

• pushBack //enqueue，将数据添加到队列的尾部（最后的位置）。

• popFront //dequee, 从队列的头部（最前的位置）移除并返回数据。

• first //peek, 返回队列头部的元素，但不移除它。

Ü:Warum spezielle Sequenz-Typen betrachten, wenn wir mit der bekannten Datenstruktur für Listen schon alle benötigten Operationen in O(1) haben?

A:

• Programme werden lesbarer und einfacher zu debuggen, wenn spezialisierte Zugriffsmuster explizit gemacht werden.可读性高，容易管理

•Einfachere Interfaces erlauben eine größere Breite von konkreten Implementationen (hier z.B. platzsparendere als Listen).简化实现的复杂度

• Listen sind ungünstig, wenn die Operationen auf dem Sekund ärspeicher (Festplatte) ausgeführt werden.

•Sequentielle Zugriffsmuster können bei entsprechender Implementation (hier z.B. als Arrays) stark vom Cache profitieren.利用缓存优化顺序读写

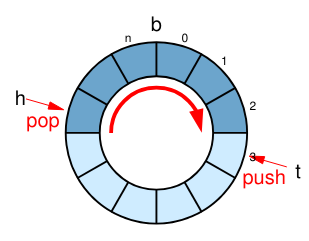
Bsp：

• Stacks mit bestimmte Größe: statische Arrays

• Stacks mit unbeschränkter Größe: dynamische Arrays Oder Stacks als einfach verkettete Listen (top of stack = front of list)

• (FIFO-)Queues: einfach verkettete Listen mit Zeiger auf letztes Element (eingefügt wird am Listenende, entnommen am Listenanfang, denn beim Entnehmen muss der Nachfolger bestimmt werden)

• Deques: doppelt verkettete Listen (einfach verkettete Listen reichen nicht)

Zirkuläre Queues

class CircularFIFO<Elem> {

const int n; // 总数

Elem[n+1] b;

int h = 0; //第一个元素

int t = 0; // 第一个空元素，也可以理解为队列末尾

}

• Queue besteht aus den Feldelementen //标注了有效元素才是Queue

h, . . . , t−1

• Es bleibt immer mindestens ein Feldelement frei (zur Unterscheidung zwischen voller und leerer Queue)类似成Dequeue，这里至少都得有一个元素，否则无法区分空还是有意义

boolean isEmpty () {return (h==t);} //如果头尾索引相同，则肯定是空的

Elem first () {assert (! isEmpty ()); return b[h];} //首先判断是否为空，其次返回有效数据b的head索引元素

int size () {return (t-h+n+1) % (n+1);} //通过计算尾指针（t）和头指针（h）之间的距离，并考虑循环队列的特性（即当尾指针超过头指针时，它们之间的距离是尾指针减去头指针加上数组的总大小再加一，然后对数组总大小加一取模），来得到队列的大小。

void pushBack (Elem x) {

assert (size () < n);

b[t] = x;

t = (t+1) % (n+1);

}

void popFront () {

assert (! isEmpty ());

h = (h+1) % (n+1);

}**ZF**：• Die Datenstruktur kann auch als Dequeue verwendet werden

• Zirkulare Arrays erlauben auch den indexierten Zugriff: ¨

Elem Operator [int i] {

return b[(h+i) % (n+1)];

}

• Statische Queues / Deques können genauso zu dynamischen Queues / Deques erweitert werden wie statische Arrays zu dynamischen Arrays

1.5 Sortierende Sequenzen

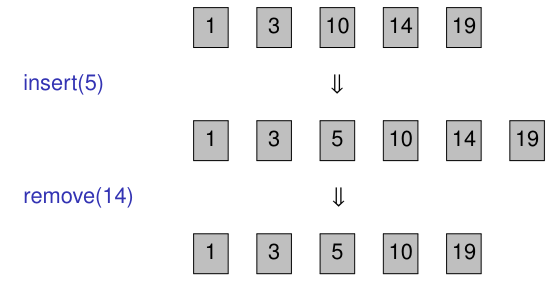
Sortierte Sequenz: Element e identifiziert über key(e)

Operationen:

• ⟨e1, . . . , en⟩.insert(e) = ⟨e1, . . . , ei, e, ei+1, . . . , en⟩ für das i mit key(ei) < key(e) < key(ei+1)

• ⟨e1, . . . , en⟩.remove(k) = ⟨e1, . . . , ei−1, ei+1, . . . , en⟩ für das i mit key(ei) = k

• ⟨e1, . . . , en⟩.find(k) = ei für das i mit key(ei) = k

Problem:: Aufrechterhaltung der Sortierung nach jeder Einfügung / Löschung每次插入/删除后保持排序

Lösung: Realisierung als Liste:

• insert und remove kosten zwar eigentlich nur konstante Zeit, müssen aber wie find

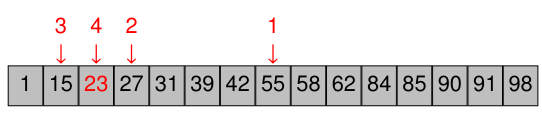
zunächst die richtige Position finden

• find auf Sequenz der Länge n kostet O(n) Zeit, damit ebenso insert und remove

Realisierung als Feld:

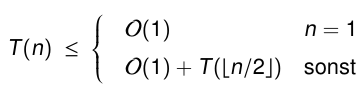
• find kann mit binärer Suche in Zeit O(log n) realisiert werden

• insert und remove kosten O(n) Zeit für das Verschieben der nachfolgenden Elemente

Bsp:find(23):

**Lemma**: In einem sortierten Feld mit n Elementen findet die binäre Suche ein beliebiges Element **in O(log n)** Laufzeit. **引理**：在具有 n 个元素的排序字段中，**二分搜索在 O(log n) 时间内找到任何元素**。

Erklärung: Suche in Feld mit n Ele

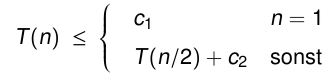


Beweis: • Master-Theorem (später)

• Raten und Induktion

Durch Induktion:

• Beschränkung auf den Fall n = 2^k (dadurch ist n/2 ganzzahlig)

• Entfernung der O-Notation:

für Konstanten c1, c2 > 0.

• “Raten”: T(n) ≤ a log n + b (also T(n) ∈ O(log n))

Induktionsinhalt:

• Induktionsanfang n = 1:

für b ≥ c1 gilt T(1) ≤ c1 ≤ a · 0 + b ✓

• Induktionsschritt n − 1 → n:

es gelte T(n′) ≤ a log n′ + b für n′ ≤ n − 1 (Induktionsvoraussetzung), dann gilt:

T(n) ≤ T(n/2) + c2

≤ a log(n/2) + b + c2 (nach IV)

= a(log n − 1) + b + c2

= a log n + (c2 − a) + b

≤ a log n + b fur¨ a ≥ c2 ✓