Verschmelzen von Sortierten Listen

Gedankenexperiment:

- 2 Personen teilen sich die Sortieraugabe
- Jeder sortiert die Hälfte der Eingabeliste
  z.b. Insertion-Sort, Bubble-Sort, . . .
- Zuletzt müssen die beiden sortierten Teillisten gemerged werden.

Algorithmus zum verschmelzen von Listen benötigt

Algorithm 1: Merge(A, l, m, r)

Data: Sorted partial lists $A[l], \ldots, A[m]$ and $A[m+1], \ldots, A[r]$
Result: Sorted list in $A[l], \ldots, A[r]$

```
i = l;  j = m + 1;  k = l;
while (i ≤ m) and (j ≤ r) do
    if (A[i] ≤ A[j]) then
        B[k] = A[i]; i = i + 1;
    else
        B[k] = A[j]; j = j + 1;
    end
    k = k + 1
end
if (i > m) then
    for h = j ... r do
        B[k + h − j] = A[h]
    end
else
    for h = i ... m do
        B[k + h − i] = A[h]
    end
end
for h = l, ..., r do
    A[h] = B[h]
end
```

Verschmelzen von Sortierten Listen

Aufwand für unser Gedankenexperiment:

- Jeder sortiert die Hälfte der Eingabeliste
  Insertion-Sort Aufwand: $2 \cdot \left(\frac{n}{2}\right)^2 = \frac{n^2}{2}$
- Verschmelzen kostet $\mathcal{O}(n)$, ist also vernachlässigbar

$\rightarrow$ doppelt so schnell wie sortieren mit Insertion-Sort

Idee: Wiederholtes Teilen der Liste, macht die Prozedur immer schneller
$\rightarrow$ Sortieren lässt sich durch *divide and conquer* lösen.

Merge-Sort

Idee: Falls Liste mehr als 1 Element hat, teile sie in der Mitte
Sortiere die zwei Teillisten *recursiv*
Verschmelze die sortierten Teillisten

Algorithm 2: Merge-Sort(A, l, r)

Data: List A, to-be-sorted Interval $l \ldots r$
Result: Sorted partial list in $A[l], \ldots, A[r]$
if $l < r$ **then**
    $m = \lfloor (l + r)/2 \rfloor$;
    Merge-Sort(A, l, m);
    Merge-Sort(A, m+1, r);
    Merge(A, l, m, r);
end

Aufwand für Merge-Sort

Der Aufwand für Merge-Sort liegt in $\mathcal{O}(n \log n)$
Intuitive Herleitung siehe Tafel ....

Formeller Beweis kann mit vollständiger Induktion geführt werden.

Ausgangspunkt: Sei $T(n)$ die Zeit um eine Liste der Länge n zu sortieren, dann gilt (mit geeigneter Konstante c):

$$T(n) = \begin{cases} c & n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lceil \frac{n}{2} \rceil) + c \cdot n & n > 1 \end{cases}$$

Aufgabe: Benutze dies um zu beweisen, dass $T(n) \leq 3c \cdot n \cdot \log_2(n)$.

Quicksort

Idee: Wähle ein Pivotelemnt x, Partitioniere die Liste, so dass Elemente $> x$s recht von x stehen, Elemente $< x$ links.

Algorithm 3: Quicksort(A, l, r)

Data: List A, to-be-sorted Interval $l \dots r$
Result: Sorted partial list in $A[l], \dots, A[r]$

```
if l < r then
    p = r;
    p = Partition(A, l, r, p);
    Quicksort(A, l, p-1);
    Quicksort(A, p, r);
end
```

Quicksort

Algorithm 4: Partition(A, l, r, p)

Data: List A, Interval $l \dots r$, pivot pos. p
Result: Pivot position i; Partitioned list $A[l], \dots, A[r]$

```
i = l − 1; j = r;
x = A[p]; swap A[p], A[r];          // park pivot in position r
repeat
    repeat
        i = i + 1;
    until A[i] ≥ x;
    repeat
        j = j − 1;
    until A[j] ≤ x or j < i;
    if i < j then
        swap A[i], A[j]
    end
until i ≥ j;
swap A[i], A[r];                    // correct position of pivot is i
return i;
```

Rechenaufwand für Partition ist *linear*.

Quicksort

Algorithm 5: Qicksort(A, l, r)

Data: List A, to-be-sorted Interval $l \dots r$
Result: Sorted partial list in $A[l], \dots, A[r]$

```
if l < r then
    p = r;
    p = Partition(A, l, r, p);
    Quicksort(A, l, p-1);
    Quicksort(A, p, r);
end
```

Vergleich Sortierverfahren

Worst case and average run time are no the only important criteria

- Stability: Equal elements are not reordered
- In place: Requires little extra space
- Nearly sorted lists should take only $\mathcal{O}(n)$ time

Sort	Advantage	Disadvantage
insertion	$\Theta(N)$ for nearly sorted	$\Theta(N^2)$ otherwise
bubble	$\Theta(N)$ for nearly sorted	$\Theta(N^2)$ otherwise
merge	$\Theta(N\ ln\ N)$	$\mathcal{O}(N)$ extra space
quick	$\Theta(N\ ln\ N)$ average	$\Theta(N^2)$ worst case, not stable

Lineare Sortierverfahren

Im Allgemeinen lässt sich Sortieren nicht schneller als $\mathcal{O}(n \log n)$ lösen.
Daten mit bekanntem (kleinen) Wertebereich (e.g. Integers in $[0, \ldots, m]$) können aber in $\mathcal{O}(n)$ sortiert werden.

Bucketsort: Verteile n Daten auf $m + 1$ “Buckets”, sammle Buckets der Reihe nach ein.

Radixsort: Für Worte der Länge l über einem fixen Alphabet: Sortiere (Bucketsort) nach erstem Buchstaben, sortiere dann jede subliste nach dem zweiten Buchsteben, etc.
Ebenso für Zahlen sortiere nach der ersten, zweiten, $\ldots$, Ziffer.

Binäre Suche

- Naive Suche in einer Liste durchläuft die Liste der Reieh nach
- Worst case (und average case) daher $\mathcal{O}(n)$.

- In sortierten Listen lässt sich viel schneller, in $\mathcal{O}(\log n)$ suchen.
- Binäre Suche halbiert das Suchinterval in jedem Schritt
- Telephonbücher wären unbrauchbar ohne sortiert zu sein.

Algorithm 6: BinarySearch(A, s, l, r)

Data: Sorted List A, search key s, Interval $l \dots r$
Result: Position of s, or 0 if not found
repeat
    $m = \lfloor (l+r)/2 \rfloor$;
    if $s < A[m]$ **then**
        $r = m - 1$;
    else
        $l = m + 1$;
    end
until $s == A[m]$ *or* $l > r$;
if $s == A[m]$ **then**
    return m;
else
    return 0;
end

Algorithm 7: InterpolationSearch(A, s, l, r)

Data: Sorted List A, search key s, Interval $l \dots r$
Result: Position of s, or 0 if not found

```
repeat
    m = ⌊(l + (r − l) (s − [A[l]) / (A[r] − A[l]))⌋;
    if s < A[m] then
        r = m − 1;
    else
        l = m + 1;
    end
until s == A[m] or l > r;
if s == A[m] then
    return m;
else
    return 0;
end
```

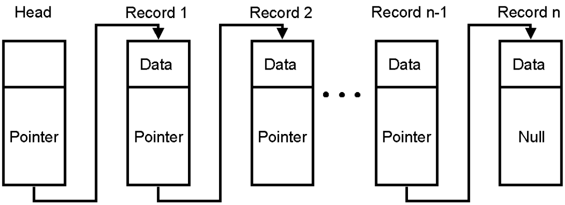
$m = \lfloor (l + (r-l)\frac{s-[A[l]]}{A[r]-A[l]}) \rfloor$

- Interpolationssuche “rät” die Position von s durch Interpolation.
- Beschleunigung für gleichmässig verteilte Daten
- Worst case aber $\mathcal{O}(n)$!

Linked Lists and Arrays

- Arrays (Felder) erlauben “random access”, i.e. direkten Zugriff auf das k-te Element
- Zum Einfügen und Löschen muss der gesamte Rest des Arrays verschoben werden
  $\rightarrow$ teure $\mathcal{O}(n)$ Operation

- Linked Lists erlauben einfaches Einfügen und Löschen ($\mathcal{O}(1)$)
- Nur sequenzieller Zugriff möglich
- random access $\mathcal{O}(n)$
- $\rightarrow$ keine binäre Suche möglich



Binäre Suchbäume

Binäre Bäume als Verallgemeinerung der linked List

$x.key$:	Schlüssel
$x.data$:	weiter Daten
$x.father$:	Zeiger auf Vater von x
$x.left$:	linkes Kind von x
$x.right$:	rechtes Kind von x

Eigenschaft binärer *Suchbaum*:
Ist y im linken Unterbaum von x, dann $y.key \leq x.key$
im rechten Unterbaum $y.key \geq x.key$

Suchen in Binärbäumen

Algorithm 8: Search(p, s)

Data: Binary tree with root p, search key s
Result: Pointer to node containing s, or NULL if not found

while $p \neq NULL$ *and* $p.key \neq s$ **do**
    if $s < p.key$ **then**
        $p = p.left$;
    else
        $p = p.right$;
    end
end
return p;

Aufwand abhängig von der Höhe des Baums $\mathcal{O}(h(T))$

Tree Traversal

Aufgabe: Durchlaufe den Baum, besuche jede Node z.b. um die Daten auszugeben

- Gewöhnlich *depth-first* nicht *breadth-first*
- Drei Möglichkeiten für depth-first:
  pre-order, in-order, post-order
- Ausgabe eines binären Baums in-order ergibt sortierte Liste

---

Algorithm 9: InOrder(p)

Data: Binary tree with root p
Result: Visit every node to do something (function Visit())
if $p \neq NULL$ **then**
    InOrder(p.left);
    Visit(p);
    Inorder(p.right);
end

Einfügen in Binären Bäumen

Einfügen einer neuen node q in den Baum

1. Suche nach $q.key$ um die richtige Stelle r zu finden
2. attache q als Kind von r
3. q wird also immer als Blatt

Aufwand wie Suche, i.e. $\mathcal{O}(h)$

Algorithm 10: Insert(root, q)

```
Data: Binary tree starting at root, new node q
r =NULL; p = root;
while p ≠NULL do
    r = p;
    if q.key < p.key then
        p = p.left;
    else
        p = p.right;
    end
end
q.father = r; q.left = NULL; q.right = NULL;
if r ==NULL then
    root = q;
else
    if q.key < r.key then
        r.left = q;
    else
        r.right = q;
    end
end
```

Successor und Predecessor

Gegeben Node q finde den nächst grösseren (kleineren) Wert

- Predecessor: Grösster Wert im linken Subbaum
- Successor: Kleinster Wert im rechten Subbaum

Algorithm 11: Successor(q)

```
if q.right ≠ NULL then
    return Minimum(q.right)
else
    p = q.father;
    while p ≠ NULL and q == p.right do
        q = p; p = p.father;
    end
    return p;
end
```

Entfernen aus Binären Bäumen

Etwas komplizierter als Einfügen, erfordert Fallunterscheidung

1. Fall q hat keine Kinder — q kann einfach entfernt werden
2. Fall q hat ein Kind — verbinde Kind von q mit dessen Vater
3. Fall q hat 2 Kinder

   Finde Ersatzknoten für q:
   Entweder Predecessor oder Successor von q
   E.g. y ist Successor von q
   Ersetze q durch y, verbinde Kind von $y.right$ mit $y.father$

Aufwand wie Suche, i.e. $\mathcal{O}(h)$

Entfernen aus Binären Bäumen

Algorithm 12: Delete(*root*, *q*)

Data: Binary tree starting at *root*, to be deleted node *q*

```
if q.left == NULL or q.right == NULL then
    r = q; // case 1 or 2: remove q itself
else
    r = Successor(q);    // case 3: remove Successor
    q.key = r.key; q.data = r.data;
end
// r is the node we remove
// now just connect child of r with father of r
// continued ...
```

```
// ... continued
// connect child p of r with father of r , in order
if r.left ≠NULL then
|   p = r.left;
else
|   p = r.right;
end
if p ≠NULL then
|   p.father = r.father
end
if r.father ==NULL then
|   root = p;    // new root p
else
|   if r == r.father.left then
|   |   r.father.left = p;
|   else
|   |   r.father.right = p
|   end
end
Free memory of node r
```

AVL Bäume

- Einfügen und Insertieren führt zu unbalancierten Bäumen
- AVL-Bäume (wie auch Red-Black Bäume) sind höhenbalancierte Bäume

- Sei $\mathit{bal}(v) = h(v.\mathit{left}) - h(v.\mathit{right})$ die Balance von v
- Ein AVL-Baum ist ein binärer Baum falls für alle Knoten v $\mathit{bal}(v) \in \{-1, 0, 1\}$
- Die Höhe eine AVL-Baums ist in $\mathcal{O}(\log n)$

Rebalancierung von AVL Bäume

- Einfügen und Insertieren führt zu unbalancierten Bäumen
- Nach jeder Insertion oder Deletion wird der Baum rebalanciert.
- Insertion und Deletion führt schlimmstenfalls zu eine Node mit $bal(v) = \pm 2$
- Es muss also nur eine kleine lokale Dis-balance ausgeglichen werden
- Dies erfolgt durch “Rotations” Operationen

$\rightarrow$ Tafel