10. Sorting III

Lower bounds for the comparison based sorting, radix- and bucket-sort

10.1 Lower bounds for comparison based sorting

[Ottman/Widmayer, Kap. 2.8, Cormen et al, Kap. 8.1]

Lower bound for sorting

Up to here: worst case sorting takes $\Omega(n \log n)$ steps.

Is there a better way? No:

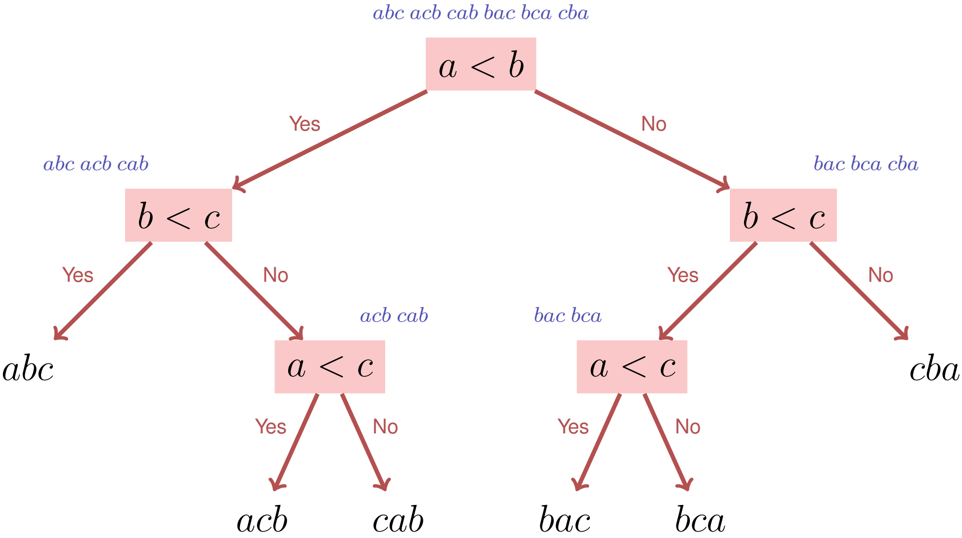
Theorem

Sorting procedures that are based on comparison require in the worst case and on average at least $\Omega(n \log n)$ key comparisons.

Comparison based sorting

- An algorithm must identify the correct one of $n!$ permutations of an array $(A_i)_{i=1,\dots,n}$.
- At the beginning the algorithm know nothing about the array structure.
- We consider the knowledge gain of the algorithm in the form of a decision tree:
  - Nodes contain the remaining possibilities.
  - Edges contain the decisions.

Decision tree



Decision tree

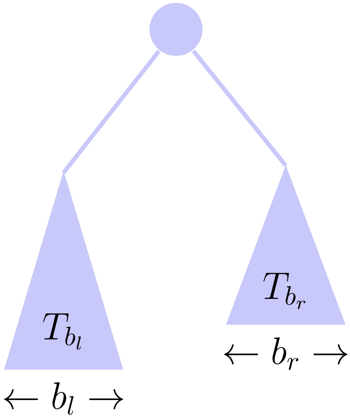
The height of a binary tree with L leaves is at least $\log_2 L$. $\Rightarrow$ The heigh of the decision tree $h \geq \log n! \in \Omega(n \log n)$.[11]

Thus the length of the longest path in the decision tree $\in \Omega(n \log n)$.

Remaining to show: mean length $M(n)$ of a path $M(n) \in \Omega(n \log n)$.

[11] $\log n! \in \Theta(n \log n)$:
$\log n! = \sum_{k=1}^{n} \log k \leq n \log n$.
$\log n! = \sum_{k=1}^{n} \log k \geq \sum_{k=n/2}^{n} \log k \geq \frac{n}{2} \cdot \log \frac{n}{2}$.

Average lower bound



- Decision tree T_n with n leaves, average height of a leaf $m(T_n)$
- Assumption $m(T_n) \geq \log n$ not for all n.
- Choose smalles b with $m(T_b) < \log n \Rightarrow b \geq 2$
- $b_l + b_r = b$, wlog $b_l > 0$ und $b_r > 0 \Rightarrow$ $b_l < b, b_r < b \Rightarrow m(T_{b_l}) \geq \log b_l$ und $m(T_{b_r}) \geq \log b_r$

Average lower bound

Average height of a leaf:

$$\begin{aligned} m(T_b) &= \frac{b_l}{b}(m(T_{b_l}) + 1) + \frac{b_r}{b}(m(T_{b_r}) + 1) \\ &\geq \frac{1}{b}(b_l(\log b_l + 1) + b_r(\log b_r + 1)) = \frac{1}{b}(b_l \log 2b_l + b_r \log 2b_r) \\ &\geq \frac{1}{b}(b \log b) = \log b. \end{aligned}$$

Contradiction. $\blacksquare$

The last inequality holds because $f(x) = x \log x$ is convex and for a convex function it holds that $f((x+y)/2) \leq 1/2 f(x) + 1/2 f(y)$ ($x = 2b_l$, $y = 2b_r$).[12]

Enter $x = 2b_l$, $y = 2b_r$, and $b_l + b_r = b$.

[12] generally $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$ for $0 \leq \lambda \leq 1$.

10.2 Radixsort and Bucketsort

Radixsort, Bucketsort [Ottman/Widmayer, Kap. 2.5, Cormen et al, Kap. 8.3]

Radix Sort

Sorting based on comparison: comparable keys ($<$ or $>$, often $=$).
No further assumptions.

Different idea: use more information about the keys.

Annahmen

Assumption: keys representable as words from an alphabet containing m elements.

Examples

$m = 10$	decimal numbers	$183 = 183_{10}$
$m = 2$	dual numbers	101_2
$m = 16$	hexadecimal numbers	$A0_{16}$
$m = 26$	words	``INFORMATIK''

m is called the radix of the representation.

Assumptions

- keys = m-adic numbers with same length.
- Procedure z for the extraction of digit k in $\mathcal{O}(1)$ steps.

Example

$$z_{10}(0, 85) = 5$$
$$z_{10}(1, 85) = 8$$
$$z_{10}(2, 85) = 0$$

Radix-Exchange-Sort

Keys with radix 2.

Observation: if $k \geq 0$,

$$z_2(i,x) = z_2(i,y) \text{ for all } i > k$$

and

$$z_2(k,x) < z_2(k,y),$$

then $x < y$.

Radix-Exchange-Sort

Idea:

- Start with a maximal k.
- Binary partition the data sets with $z_2(k, \cdot) = 0$ vs. $z_2(k, \cdot) = 1$ like with quicksort.
- $k \leftarrow k - 1$.

Radix-Exchange-Sort

0111 0110 1000 0011 0001

0111 0110 0001 0011 | 1000

0011 0001 | 0110 0111 | 1000

0001 | 0011 | 0110 0111 | 1000

0001 | 0011 | 0110 | 0111 | 1000

Algorithm RadixExchangeSort(A, l, r, b)

Input : Array A with length n, left and right bounds $1 \leq l \leq r \leq n$, bit position b
Output : Array A, sorted in the domain $[l, r]$ by bits $[0, \ldots, b]$.

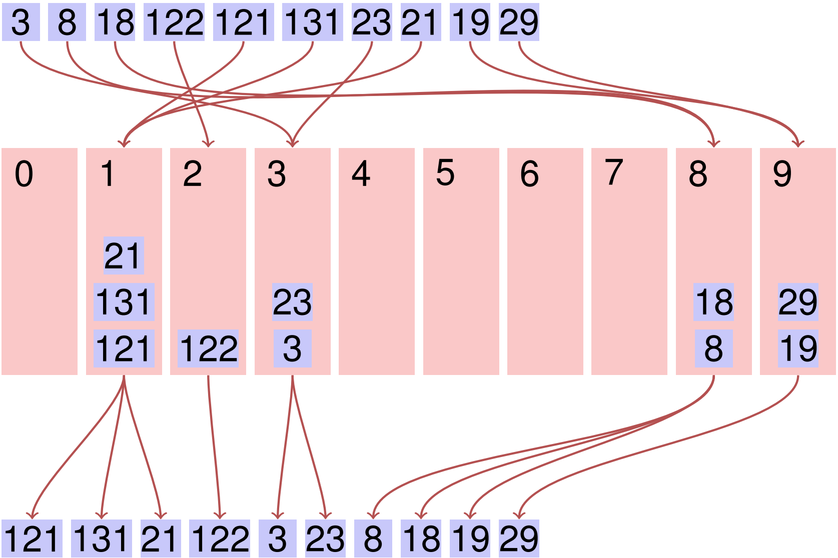
if $l > r$ **and** $b \geq 0$ **then**
    $i \leftarrow l - 1$
    $j \leftarrow r + 1$
    repeat
        repeat $i \leftarrow i + 1$ **until** $z_2(b, A[i]) = 1$ **and** $i \geq j$
        repeat $j \leftarrow j + 1$ **until** $z_2(b, A[j]) = 0$ **and** $i \geq j$
        if $i < j$ **then** swap($A[i], A[j]$)
    until $i \geq j$
    RadixExchangeSort($A, l, i - 1, b - 1$)
    RadixExchangeSort($A, i, r, b - 1$)

Analysis

RadixExchangeSort provide recursion with maximal recursion depth = maximal number of digits p.

Worst case run time $\mathcal{O}(p \cdot n)$.

Bucket Sort



3 8 18 122 121 131 23 21 19 29
0
1
2
3
4
5
6
7
8
9
21
131
121
122
23
3
18
8
29
19
121 131 21 122 3 23 8 18 19 29

Bucket Sort

121 131 21 122 3 23 8 18 19 29

0	1	2	3	4	5	6	7	8	9
		29							
		23							
		122							
8	19	21							
3	18	121	131						

3 8 18 19 121 21 122 23 29

Bucket Sort

3 8 18 19 121 21 122 23 29

0	1	2	3	4	5	6	7	8	9
29									
23									
21									
19									
18	131								
8	122								
3	121								

3 8 18 19 21 23 29 121 122 131

implementation details

Bucket size varies greatly. Two possibilities

- Linked list for each digit.
- One array of length n. compute offsets for each digit in the first iteration.

11. Fundamental Data Types

Abstract data types stack, queue, implementation variants for linked lists, amortized analysis [Ottman/Widmayer, Kap. 1.5.1-1.5.2, Cormen et al, Kap. 10.1.-10.2,17.1-17.3]

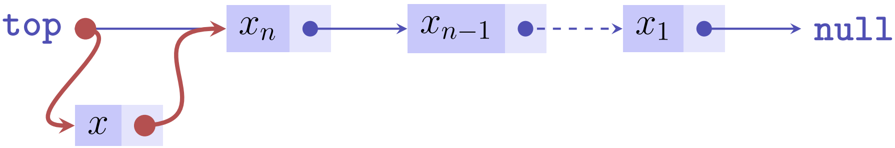
Abstract Data Types

We recall

A *stack* is an abstract data type (ADR) with operations

- `push`(x, S): Puts element x on the stack S.
- `pop`(S): Removes and returns top most element of S or `null`
- `top`(S): Returns top most element of S or `null`.
- `isEmpty`(S): Returns `true` if stack is empty, `false` otherwise.
- `emptyStack`$()$: Returns an empty stack.

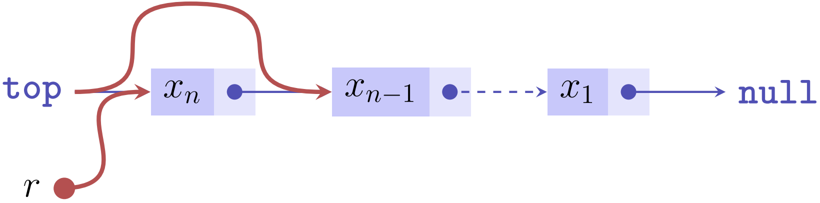
Implementation Push



push(x, S):

1. Create new list element with x and pointer to the value of **top**.
2. Assign the node with x to **top**.

Implementation Pop



$\texttt{pop}(S)$:

1. If $\texttt{top}$=$\texttt{null}$, then return $\texttt{null}$
2. otherwise memorize pointer p of $\texttt{top}$ in r.
3. Set $\texttt{top}$ to $p.next$ and return r

Analysis

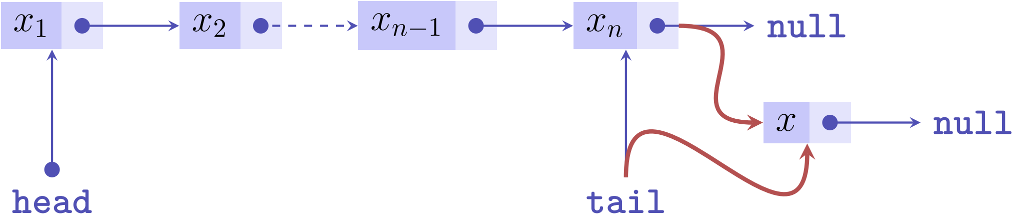
Each of the operations `push`, `pop`, `top` and `isEmpty` on a stack can be executed in $\mathcal{O}(1)$ steps.

Queue (fifo)

A queue is an ADT with the following operations

- `enqueue`(x, Q): adds x to the tail (=end) of the queue.
- `dequeue`(Q): removes x from the head of the queue and returns x (`null` otherwise)
- `head`(Q): returns the object from the head of the queue (`null` otherwise)
- `isEmpty`(Q): return `true` if the queue is empty, otherwise `false`
- `emptyQueue`$()$: returns empty queue.

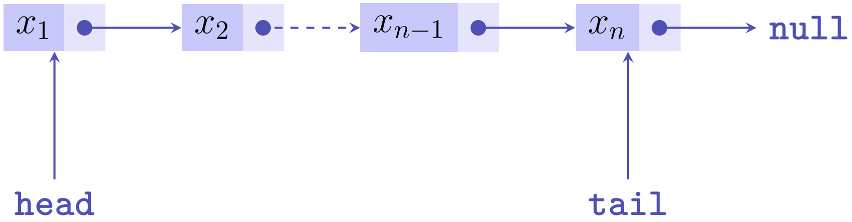
Implementation Queue



`enqueue`(x, S):

1. Create a new list element with x and pointer to `null`.
2. If `tail` $\neq$ `null`, then set `tail.next` to the node with x.
3. Set `tail` to the node with x.
4. If `head` $=$ `null`, then set `head` to `tail`.

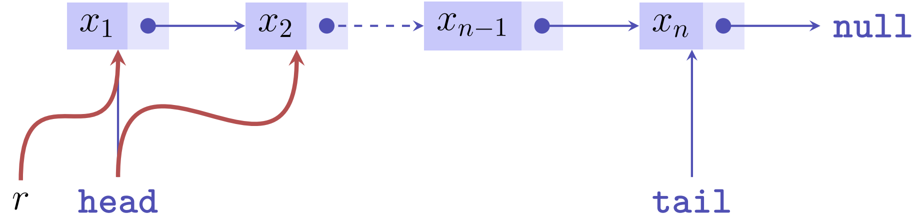
Invariants



With this implementation it holds that

- either $\texttt{head} = \texttt{tail} = \texttt{null}$,
- or $\texttt{head} = \texttt{tail} \neq \texttt{null}$ and $\texttt{head.next} = \texttt{null}$
- or $\texttt{head} \neq \texttt{null}$ and $\texttt{tail} \neq \texttt{null}$ and $\texttt{head} \neq \texttt{tail}$ and $\texttt{head.next} \neq \texttt{null}$.

Implementation Queue



`dequeue`(S):

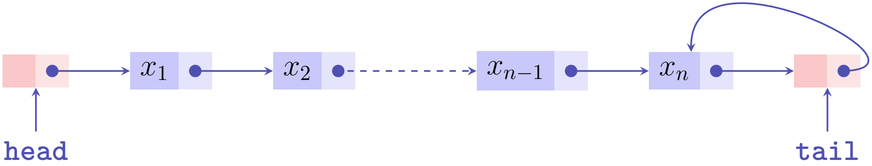
1. Store pointer to `head` in r. If $r =$ `null`, then return r .
2. Set the pointer of `head` to `head.next`.
3. Is now `head` $=$ `null` then set tail to `null`.
4. Return the value of r.

Analysis

Each of the operations `enqueue`, `dequeue`, `head` and `isEmpty` on the queue can be executed in $\mathcal{O}(1)$ steps.

Implementation Variants of Linked Lists

List with dummy elements (sentinels).

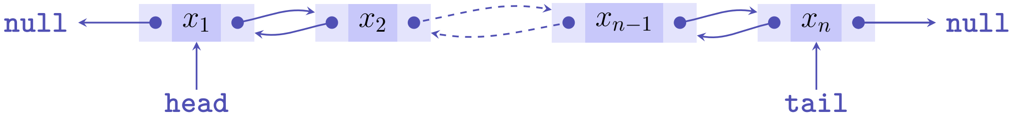


Advantage: less special cases

Variant: like this with pointer of an element stored singly indirect.

Implementation Variants of Linked Lists

Doubly linked list



Overview

	enqueue	insert	delete	search	concat
(A)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
(B)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
(C)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
(D)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

(A) = singly linked
(B) = Singly linked with dummy
(C) = Singly linked with indirect element addressing
(D) = doubly linked

priority queue

Priority Queue

Operations

- `insert(x,p,Q)`: Enter object x with priority p.
- `extractMax(Q)`: Remove and return object x with highest priority.

Implementation Priority Queue

With a Max Heap

Thus

- `insert` in $\mathcal{O}(\log n)$ and
- `extractMax` in $\mathcal{O}(\log n)$.

Multistack

Multistack adds to the stack operations below

`multipop(s,S)`: remove the $\min(\text{size}(S), k)$ most recently inserted objects and return them.

Implementation as with the stack. Runtime of `multipop` is $\mathcal{O}(k)$.

Academic Question

If we execute on a stack with n elements a number of n times `multipop(k,S)` then this costs $\mathcal{O}(n^2)$?

Certainly correct because each `multipop` may take $\mathcal{O}(n)$ steps.

How to make a better estimation?

Idea (accounting)

Introduction of a cost model:

- Each call of `push` costs 1 CHF and additional 1 CHF will be put to account.
- Each call to `pop` costs 1 CHF and will be paid from the account.

Account will never have a negative balance. Thus: maximal costs = number of `push` operations times two.

More Formal

Let t_i denote the real costs of the operation i. Potential function $\Phi_i \geq 0$ for the "account balance" after i operations. $\Phi_i \geq \Phi_0 \; \forall i$.

Amortized costs of the ith operation:

$$a_i := t_i + \Phi_i - \Phi_{i-1}.$$

It holds

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{n} t_i \right) + \Phi_n - \Phi_0 \geq \sum_{i=1}^{n} t_i.$$

Goal: find potential function that evens out expensive operations.

Example stack

Potential function Φ_i = number element on the stack.

- `push`(x, S): real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = 1$. Amortized costs $a_i = 2$.
- `pop`(S): real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = -1$. Amortized costs $a_i = 0$.
- `multipop`(k, S): real costs $t_i = k$. $\Phi_i - \Phi_{i-1} = -k$. amortized costs $a_i = 0$.

All operations have *constant amortized cost*! Therefore, on average Multipop requires a constant amount of time.

Example Binary Counter

Binary counter with k bits. In the worst case for each count operation maximally k bitflips. Thus $\mathcal{O}(n \cdot k)$ bitflips for counting from 1 to n. Better estimation?

Real costs t_i = number bit flips from 0 to 1 plus number of bit-flips from 1 to 0.

$$...0\underbrace{1111111}_{l \text{ Einsen}}+1 = ...1\underbrace{0000000}_{l \text{ Zeroes}}.$$

$$\Rightarrow t_i = l + 1$$

Example Binary Counter

$$\ldots 0\underbrace{1111111}_{l \text{ Einsen}}+1 = \ldots 1\underbrace{0000000}_{l \text{ Nullen}}$$

potential function Φ_i: number of 1-bits of x_i.

$$\Rightarrow \Phi_i - \Phi_{i-1} = 1 - l,$$

$$\Rightarrow a_i = t_i + \Phi_i - \Phi_{i-1} = l + 1 + (1 - l) = 2.$$

Amortized constant cost for each count operation. ☺