COMP20007 Design of Algorithms

Sorting - Part 1

Daniel Beck

Lecture 14

Semester 1, 2023

Mergesort

function $\text{MERGESORT}(A[0..n-1])$
 if $n > 1$ **then**
  $B[0..\lfloor n/2 \rfloor - 1] \leftarrow A[0..\lfloor n/2 \rfloor - 1]$
  $C[0..\lceil n/2 \rceil - 1] \leftarrow A[\lfloor n/2 \rfloor..n-1]$
  $\text{MERGESORT}(B[0..\lfloor n/2 \rfloor - 1])$
  $\text{MERGESORT}(C[0..\lceil n/2 \rceil - 1])$
  $\text{MERGE}(B, C, A)$

Mergesort - Merge function

function $\textsc{Merge}(B[0..p-1], C[0..q-1], A[0..p+q-1])$

$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$

while $i < p$ **and** $j < q$ **do**

if $B[i] \leq C[j]$ **then**

$A[k] \leftarrow B[i]; i \leftarrow i + 1$

else

$A[k] \leftarrow C[j]; j \leftarrow j + 1$

$k \leftarrow k + 1$

if $i = p$ **then**

$A[k..p+q-1] \leftarrow C[j..q-1]$

else

$A[k..p+q-1] \leftarrow B[i..p-1]$

Mergesort - Properties

Two important properties of sorting algorithms:

- Being in-place: whether operations happen within the input array **or** the algorithm requires additional memory.
- Stability: if two elements are equal, algorithm preserves their original order.

For Mergesort:

- **Not** in-place: requires $\Theta(n)$ auxiliary array.
- **Stable**: Merge keeps relative order with additional bookkeeping.

Mergesort - Complexity

Divide-and-Conquer algorithm: potential for Master Theorem application.

- Worst case?
- Best case?
- Average case?

Workshop exercise.

Mergesort - In Practice

- Guaranteed $\Theta(n \log n)$ complexity
- Highly parallelisable
- Multiway Mergesort: excellent for secondary memory
- Used in JavaScript (Mozilla)
- Basis for hybrid algorithms (TimSort: Python, Android)

Take-home message: Mergesort is an excellent choice if stability is required and the extra memory cost is low.

Quicksort

function $\textsc{Quicksort}(A[l..r])$ $\triangleright$ Starts with $A[0..n-1]$

  if $l < r$ **then**

    $s \leftarrow \textsc{Partition}(A[l..r])$

    $\textsc{Quicksort}(A[l..s-1])$

    $\textsc{Quicksort}(A[s+1..r])$

Quicksort - Lomuto partitioning

function $\textsc{LomutoPartition}(A[l..r])$

    $p \leftarrow A[l]$

    $s \leftarrow l$

    for $i \leftarrow l + 1$ to r **do**

        if $A[i] < p$ **then**

            $s \leftarrow s + 1$

            $\textsc{Swap}(A[s], A[i])$

    $\textsc{Swap}(A[l], A[s])$

    return s

Quicksort - Properties

- **In-place**. (still requires $O(\log n)$ memory for the stack)
- **Not** Stable. Arbitrary element swaps break stability.

Quicksort - Partitioning

- Lomuto partitioning can be used but not the best in pratice.
- Instead, practical implementations use Hoare partitioning (proposed by the inventor of Quicksort).
- How does it work? Let's go back to my cards first...

Quicksort - Hoare partitioning

function $\textsc{HoarePartition}(A[l..r])$

    $p \leftarrow A[l]$

    $i \leftarrow l; j \leftarrow r + 1$

    repeat

        repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$

        repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$

        $\textsc{Swap}(A[i], A[j])$

    until $i \geq j$

    $\textsc{Swap}(A[i], A[j])$

    $\textsc{Swap}(A[l], A[j])$

    return j

Quicksort - Complexity

Another Divide-and-Conquer algorithm.

- Worst case?
- Best case?
- Average case?

Warning: average case analysis is not trivial (non-examinable).

Quicksort - Best Case Complexity

Quicksort - Worst Case Complexity

Quicksort - Average Case Complexity

Quicksort - In practice

- Used in C (qsort)
- Basis for C++ sort (Introsort)
- **Fastest** sorting algorithm in most cases

Take-home message: Quicksort is the algorithm of choice when speed matters and stability is not required.

Summary so far

Selection Sort: Slow, $\Theta(n^2)$ in all cases. But only $O(n)$ key exchanges.

Mergesort: Better for mid-size arrays and when stability is required.

Quicksort: Usually the best choice for large arrays, with excellent **empirical** performance.

Next lecture: Heapsort

COMP20007 Design of Algorithms

Sorting - Part 2

Daniel Beck

Lecture 15

Semester 1, 2023

Priority Queues

- A *set* of elements, each one with a *priority key*.
- **Inject:** put a new element in the queue.
- **Eject:** find the element with the *highest* priority and remove it from the queue.
- Used as part of an algorithm (ex: Dijkstra)...
- ...or on its own (ex: OS job scheduling).

Sorting using a Priority Queue

- Different implementations result in different sorting algorithms.
- If using an *unsorted array/list*, we obtain Selection Sort.
- If using an *heap*, we obtain Heapsort.

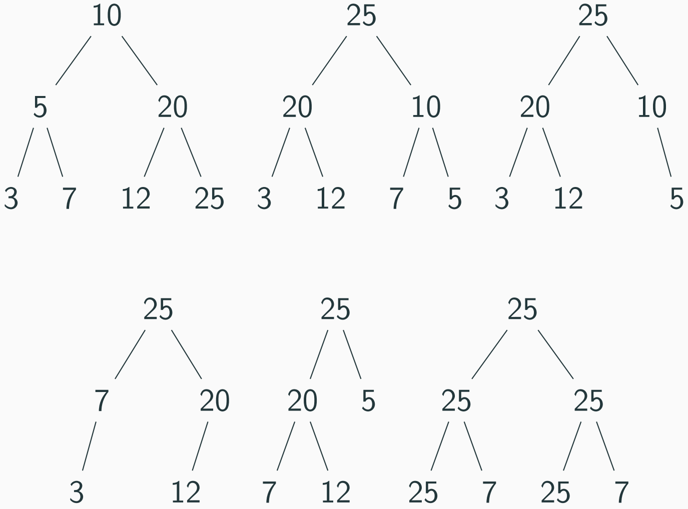
The Heap

It's a tree with a set properties:

- Binary (at most two children allowed per node)
- Complete (all levels are full except for the last, where only rightmost leaves can be missing)
- Parental dominance (the key of a parent node is always higher than the key of its children)

Heaps and Non-Heaps

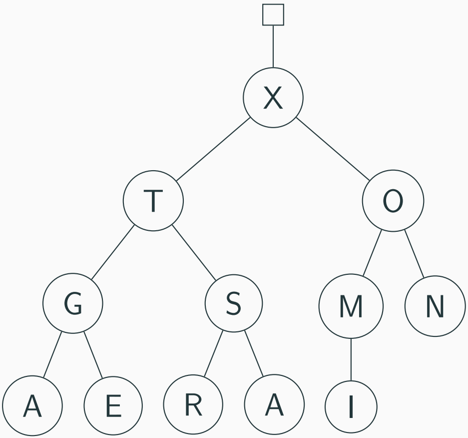
Which of these are heaps?



Heaps as Arrays

We can utilise the completeness of the tree and place its elements in level-order in an array A.

Note that the children of node i will be nodes $2i$ and $2i+1$.

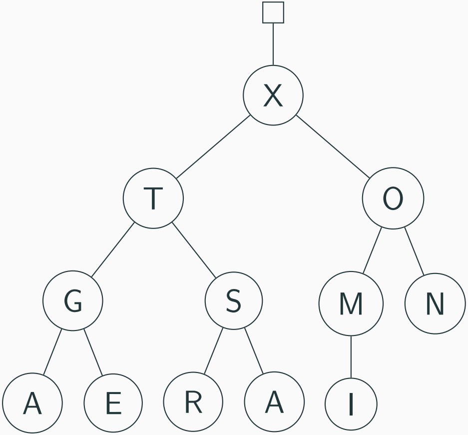


	X	T	O	G	S	M	N	A	E	R	A	I
0	1	2	3	4	5	6	7	8	9	10	11	12

Heaps as Arrays

This way, the heap condition is simple:

$\forall i \in \{0, 1, \ldots, n\}$, we must have $A[i] \leq A[\lfloor i/2 \rfloor]$.



	X	T	O	G	S	M	N	A	E	R	A	I
0	1	2	3	4	5	6	7	8	9	10	11	12

Heapsort

function $\textsc{Heapsort}(A[1..n])$ $\triangleright$ Assume $A[0]$ as a sentinel
    $\textsc{Heapify}(A[1..n])$
    for $i \leftarrow n$ to 0 **do**
        $\textsc{Eject}(A[1..i])$

Heapify

function $\textsc{BottomUpHeapify}(A[1..n])$
  for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 **do**
    $k \leftarrow i$
    $v \leftarrow A[k]$
    $heap \leftarrow False$
    while not $heap$ **and** $2 \times k \leq n$ **do**
      $j \leftarrow 2 \times k$
      if $j < n$ **then** ▷ two children
        if $A[j] < A[j+1]$ **then** $j \leftarrow j + 1$
      if $v \geq A[j]$ **then** $heap \leftarrow True$
      else $A[k] \leftarrow A[j]$; $k \leftarrow j$
    $A[k] \leftarrow v$

Eject

function $\text{EJECT}(A[1..i])$

  $\text{SWAP}(A[i], A[1])$

  $k \leftarrow 1$

  $v \leftarrow A[k]$

  $heap \leftarrow False$

  while not $heap$ **and** $2 \times k \leq i - 1$ **do**

    $j \leftarrow 2 \times k$

    if $j < i - 1$ **then** ▷ two children

      if $A[j] < A[j+1]$ **then** $j \leftarrow j + 1$

    if $v \geq A[j]$ **then** $heap \leftarrow True$

    else $A[k] \leftarrow A[j]$; $k \leftarrow j$

  $A[k] \leftarrow v$

Heapsort - Properties

Transform-and-Conquer paradigm

- Exactly as Selection Sort, but with a *preprocessing* step.
- **In-place**
- **Not** stable

Heapsort - Complexity

- Top-Down Heapify: $O(n \log n)$
- Bottom-Up Heapify: $O(n)$
- Heapsort: $C_{heapify}(n) + (n \times C_{eject}(n))$

$O(n \log n)$

Heapsort - Complexity

Heapsort - Complexity (best case)

Heapsort - In practice

- Vs. Mergesort: fully in-place but not stable
- Vs. Quicksort: guaranteed $\Theta(n \log n)$ performance but empirically slower.
- Used in the Linux kernel.

Take-home message: Heapsort is the best choice when low-memory footprint is required and guaranteed $O(n \log n)$ performance is needed (for example, security reasons).

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.
- Mergesort: good for mid-size data and when stability is required. Also good with secondary memory devices. Extra $O(n)$ memory cost can be a barrier.
- Quicksort: best for more general cases and large amounts of data. Not stable.
- Heapsort: slower in practice but low-memory and guaranteed $O(n \log n)$ performance.

Next lecture: distribution sorting.

COMP20007 Design of Algorithms

Sorting - Part 3

Daniel Beck

Lecture 16

Semester 1, 2023

Simple Distribution Sort

1 4 0 6 5 3 2

Looks $\Theta(n)$ even in worst case! Is it really?

Simple Distribution Sort

10 41 02 64 53 39 27

$\Theta(n + k)$ worst case.

Simple Distribution Sort

10 41 02 10 41 10 10

Use the auxiliary array to store counts.

Counting Sort

6 3 3 8 1 0 8 7 9 2 5 3 5 3 1 8 7 6 5 1 2 1 5 3

Key	0	1	2	3	4	5	6	7	8	9
Counts	1	4	2	5	0	4	2	2	3	1
Dist	1	5	7	12	12	16	18	20	23	24

6 3 3 8 1 0 8 7 9 2 5 3 5 3 1 8 7 6 5 1 2 1 5 3

0 1 1 1 1 2 2 3 3 3 3 3 5 5 5 5 6 6 7 7 8 8 8 9

Counting Sort

function $\text{COUNTING SORT}(A[0..n-1], l, u)$ $\triangleright$ l, u are lower and upper bounds on the keys

for $j \leftarrow 0$ **to** $u - l$ **do**

$D[j] \leftarrow 0$

for $i \leftarrow 0$ **to** $n - 1$ **do**

$D[A[i] - l] \leftarrow D[A[i] - l] + 1$

for $j \leftarrow 1$ **to** $u - l$ **do**

$D[j] = D[j] + D[j - 1]$

for $i \leftarrow n - 1$ **down to** 0 **do**

$j \leftarrow A[i] - l$

$S[D[j] - 1]] \leftarrow A[i]$

$D[j] \leftarrow D[j] - 1$

return $S[0..n-1]$

Counting Sort

- **Stable**
- **Not** In-place: needs an output array, no swaps

Take-home message: Counting Sort only works for integer keys and it works best when the key range is small.

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110	011	011	001	000	111	010	101	011	101	011	001	111	110	101	001	010	001	101	011

110	000	010	110	010															

110	000	010	110	010	011	011	001	111	101	011	101	011	001	111	101	001	001	101	011

000	001	101	101	001	101	001	001	101											

000	001	101	101	001	101	001	001	101	110	010	110	010	011	011	111	011	011	111	011

000	001	001	001	001	010	010	011	011	011	011									

000	001	001	001	001	010	010	011	011	011	011	011	101	101	101	101	110	110	111	111

0 1 1 1 1 2 2 3 3 3 3 3 5 5 5 5 6 6 7 7

Radix Sort

Assumptions:

- Maximum key length is known in advance.
- Keys can be sorted in **lexicographical** order (strings).

Start sorting from least to the most significant digit.
Total worst case performance is $\Theta(n \times len(k))$

Radix Sort

function $\textsc{Radix Sort}(A[0..n-1], k)$
  for $j \leftarrow 0$ **to** $len(k)$ **do**
    $A \leftarrow \textsc{AuxSort}(A, k[j])$

- Typically, $\textsc{AuxSort}$ is Counting Sort. But can be any sorting algorithm as long as it is stable.

Take-home message: Radix Sort can be very fast (faster than comparison sorting) if keys are short (need to known in advance).

Summary

- Distribution Sorting is a paradigm that rely on more assumptions, unlike Comparison Sorting algorithms:
    - Counting Sort: positive integer keys, with max bound known.
    - Radix Sort: more general but max key length must be known and keys should have lexicographical order.

In Practice

- Distribution Sort is not as widely used as Comparison Sort:
    - Less general.
    - In practice, good sorting algorithms can be very close to linear performance (Timsort).
- However, they can be very useful as part of a more complex algorithm.
    - Radix Sort is used to construct suffix arrays.
    - Controlled environment with guaranteed short key size.

Next lecture: dictionaries.