Algorithms & Data Structures

Fabian Bosshard

June 8, 2025

Contents

1 Introduction

$T(n) :=$ number of **basic steps** needed to compute the result of a problem of size n

Example 1. How many $1, 2$-beats can one compose over a total of n beats?

Idea: the number of $1, 2$-beats over n beats is the sum of the number of $1, 2$-beats over $n-1$ beats and the number of $1, 2$-beats over $n-2$ beats

Algorithm 1 Pingala

```
function Pingala(n)                    ▷ Number of 1,2-beats in n beats
    if n ≤ 2 then
        return n
    return Pingala(n − 1) + Pingala(n − 2)
```

$$T(1) = 1, \quad T(2) = 1, \quad T(n) = T(n-1) + T(n-2) + 2$$

$$T(n) \geq \underbrace{T(n-1)}_{\geq T(n-2)} + T(n-2) \geq 2T(n-2)$$

$$T(n) \geq 2T(n-2) \geq 2(2T(n-4)) = 2^2 T(n-4) \geq \ldots \geq 2^{n/2} T(0) = \sqrt{2}^n T(0)$$ ◀

2 Basics

Definition 1. We define the following families of functions:

- $O(g(n)) := \{f(n) \mid \exists c > 0, n_0 \in \mathbb{N} \mid 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$
- $\Omega(g(n)) := \{f(n) \mid \exists c > 0, n_0 \in \mathbb{N} \mid 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$
- $\Theta(g(n)) := \{f(n) \mid \exists c_1, c_2 > 0, n_0 \in \mathbb{N} \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \; \forall n \geq n_0\}$
- $o(g(n)) := \{f(n) \mid \forall c > 0, \exists n_0 \in \mathbb{N} \mid 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$
- $\omega(g(n)) := \{f(n) \mid \forall c > 0, \exists n_0 \in \mathbb{N} \mid 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$

The notation '$f(n) =$' is used to denote that $f(n)$ is an element of the set of functions on the right-hand side.

Example 2. Let $\pi(n)$ be the number of primes less than or equal to n. Then

$$\pi(n) = \Theta\left(\frac{n}{\log n}\right).$$ ◀

The Θ-notation, Ω-notation, and O-notation can be viewed as the "asymptotic" $=$, $\geq$, and $\leq$ relations for functions. The o-notation and ω-notation can be viewed as asymptotic $<$ and $>$.

Theorem 1. $f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)) \Leftrightarrow f(n) = \Theta(g(n))$ ◁

The above theorem can be interpreted as saying

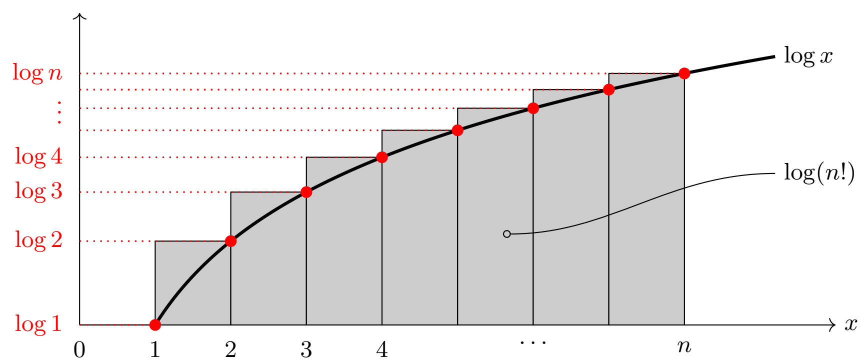
$$f \geq g \wedge f \leq g \Leftrightarrow f = g$$

Example 3. $\log(n!) \in \Theta(n \log n)$. We can rewrite $\log(n!)$ as

$$\log(n!) = \log\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log i \tag{1}$$

Clearly, $\log(n!) \in O(n \log n)$, since $n \log n = \log(n^n) = \sum_{i=1}^{n} \log n \geq \sum_{i=1}^{n} \log i$.

One way to understand why $\log(n!) \in \Omega(n \log n)$ is to interpret (1) as a sum of areas, each rectangle has width 1 and height $\log i$:



We can see that the area of the gray rectangles is bounded from below by the area under the curve $y = \log x$, i.e.

$$\int_1^n \log x \, \mathrm{d}x \leq \log(n!) \tag{2}$$

Using integration by parts, we can express the left-hand side of (2) as

$$\int_1^n \log x \, \mathrm{d}x = \int_1^n \underset{\uparrow}{1} \cdot \underset{\downarrow}{\log x} \, \mathrm{d}x = \left[x \log x - \int x \cdot \frac{1}{x} \, \mathrm{d}x\right]_1^n = [x \log x - x]_1^n$$
$$= n \log n - n + 1 \in \Omega(n \log n)$$

Therefore, $\log(n!) \in \Omega(n \log n) \wedge \log(n!) \in O(n \log n)$. Using Theorem 1, we conclude that $\log(n!) \in \Theta(n \log n)$. ◀

When $f(n) = O(g(n))$, we say that $g(n)$ is an **upper bound** for $f(n)$, and that $g(n)$ **dominates** $f(n)$.

When $f(n) = \Omega(g(n))$, we say that $g(n)$ is a **lower bound** for $f(n)$.

When $f(n) = \Theta(g(n))$, we say that $g(n)$ is a **tight bound** for $f(n)$.

We use the o-notation to denote an upper bound that is not asymptotically tight, and the ω-notation to denote a lower bound that is not asymptotically tight. The following two implications hold:

$$f(n) = o(g(n)) \Rightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \qquad f(n) = \omega(g(n)) \Rightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Notation	Name	Example
$O(1)$	constant	Finding median of *sorted* array; computing $(-1)^n$; using a fixed-size lookup table.
$O(\alpha(n))$	inverse Ackermann	Amortized cost per operation in a disjoint-set data structure.
$O(\log^* n)$	iterated logarithmic	Distributed coloring of cycles (Cole-Vishkin algorithm).
$O(\log \log n)$	double logarithmic	Interpolation search on uniformly distributed data.
$O(\log n)$	logarithmic	Binary search; operations in balanced trees or a binomial heap.
$O(\log^c n)$, $c > 1$	polylogarithmic	Matrix-chain ordering on a PRAM.
$O(n^c)$, $0 < c < 1$	fractional power	Searching in a k-d tree.
$O(\frac{n}{\log n})$		#Primes $\leq n$ (Example 1).
$O(n)$	linear	Scanning an unsorted list; ripple-carry addition of two n-bit integers.
$O(n \log^* n)$		Seidel's polygon-triangulation algorithm.
$O(n \log n) = O(\log n!)$	linearithmic	Fastest comparison sorts; Fast Fourier transform.
$O(n^2)$	quadratic	Schoolbook multiplication; bubble/selection/insertion sort; worst-case quicksort; direct convolution.
$O(n^3)$	cubic	Naive $n \times n$ matrix multiplication; partial correlation.
$O(n^c)$, $c > 1$	polynomial	TAG parsing; bipartite matching; determinant via LU.
$L_n[\alpha, c]$	sub-exponential	Factoring via number-field sieve.
$O(c^n)$, $c > 1$	exponential	Exact TSP by DP; brute-force logical equivalence checking.
$O(n!)$	factorial	Brute-force TSP; enumerating permutations or partitions; determinant by Laplace expansion.

2.1 Incremental Algorithms

Example 4 (Hand of cards)**.** Insertion sort (Algorithm 3) uses an algorithm design technique called **incremental** method: for each element $A[i]$, it inserts it into its proper place in the subarray $A[1:i]$, having already sorted $A[1:i-1]$. This is reminiscent of how one might sort a hand of cards, where you pick up a card and insert it into the correct position in the already sorted hand.

At the start of each iteration of the for loop, the subarray $A[1:i-1]$ consists of the elements originally in $A[1:i-1]$, but in sorted order. This is a loop invariant (Section 2.2). ◂

2.2 Loop Invariant

When using a **loop invariante**, 3 things need to be shown:

1. **Initialization:** It is true prior to the first iteration of the loop.
2. **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
3. **Termination:** When the loop terminates, the invariant gives a useful property that helps show that the algorithm is correct.

A loop-invariant proof is a form of **mathematical induction**, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step. The third property is perhaps the most important one, since you are using the loop invariant to show correctness. Typically, you use the loop invariant along with the condition that caused the loop to terminate. Mathematical induction typically applies the inductive step infinitely, but in a loop invariant the "induction" stops when the loop terminates.

2.3 Correctness

You are given a problem P and an algorithm A. P formally defines a **correctness** condition. Assume, for simplicity, that A consists of one loop.

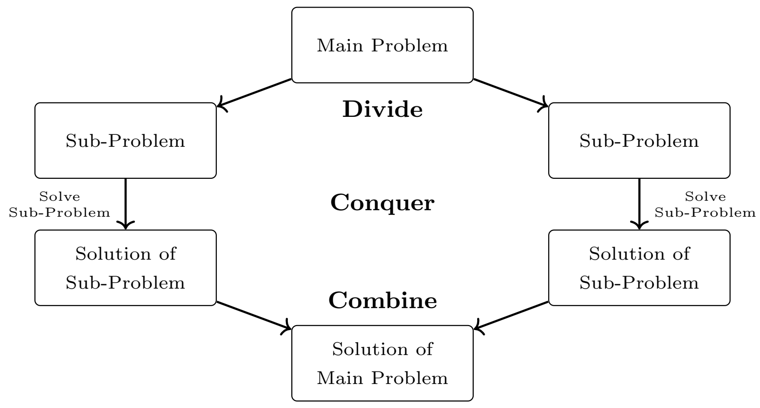
1. Formulate an invariant C
2. **Initialization**: prove that C holds right before the first execution of the first instruction of the loop
3. **Management**: prove that if C holds right before the first instruction of the loop, then it holds also at the end of the loop
4. **Termination**: prove that the loop terminates, with some exit condition X
5. Prove that $X \wedge C \Rightarrow P$, which means that A is correct

2.4 Divide-and-Conquer Algorithms

Many useful algorithms are **recursive**: they **recurse** (call themselves) one or more times to handle closely related subproblems. These algorithms typically follow the **divide-and-conquer** method: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

In the divide-and-conquer method, if the problem is small enough (the **base case**), you just solve it directly without recursing. Otherwise (the **recursive case**), you perform three characteristic steps:

1. **Divide** the problem into one or more subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the subproblem solutions to form a solution to the original problem.



When an algorithm contains a recursive call, we can often describe its running time with a **recurrence relation**, which expresses the overall running time of a problem of size n in terms of the running time of the same algorithm on smaller inputs.

2.5 Binary Search

Algorithm 2 is an efficient method for finding an element x in a sorted array A. By repeatedly halving the search interval, it reduces the problem size exponentially: at each step, it compares x to the middle element of the current interval and discards the half in which x cannot lie. This yields a worst-case running time of $O(\log n)$, a dramatic improvement over a linear search's $O(n)$ behavior.

Algorithm 2 Binary Search

```
function BinarySearch(A, x)
    l ← 1                                   ▷ leftmost index
    r ← len(A)                              ▷ rightmost index
    while l ≤ r do
        m ← ⌊(l + r)/2⌋                     ▷ midpoint of A[l : r]
        if A[m] < x then
            l ← m + 1                       ▷ search in right half
        else if A[m] > x then
            r ← m − 1                       ▷ search in left half
        else
            return m                        ▷ found x at index m
    return 0                                ▷ not found: 0 is not a valid index
```

3 Sorting

3.1 Insertion Sort

Was already used to sort cards in Section 2.1, Example 4.

Algorithm 3 Insertion Sort

```
function InsertionSort(A)
    for i = 2, ..., len(A) do
        j ← i
        while j > 1 ∧ A[j − 1] > A[j] do
            swap A[j] and A[j − 1]
            j ← j − 1
```

3.2 Merge Sort

Algorithm 4 Merge Sort

```
function MergeSort(A)
    if len(A) ≤ 1 then                      ▷ base case (array is trivially sorted)
        return A
    m = ⌊len(A)/2⌋                          ▷ midpoint of A
    A_L ← MergeSort(A[1 : m])               ▷ recursively sort A[1 : m]
    A_R ← MergeSort(A[m+1 : |A|])           ▷ recursively sort A[m+1 : |A|]
    return Merge(A_L, A_R)                  ▷ merge the two sorted arrays
```

Algorithm 5 Merge

```
function Merge(A, B)
    i, j ← 1
    C ← []
    while i ≤ len(A) ∨ j ≤ len(B) do
        if i ≤ len(A) ∧ (j > len(B) ∨ A[i] < B[j]) then
            append A[i] to C
            i ← i + 1
        else
            append B[j] to C
            j ← j + 1
    return C
```

We describe the running time of merge sort (Algorithm 4) as follows:

1. **Divide:** compute the middle of the array (Algorithm 4, Line 4), which takes constant time, $D(n) = \Theta(1)$
2. **Conquer:** recursively solve two subproblems (Algorithm 4, Line 5, 6), each of size $n/2$, contributes $2T(n/2)$
3. **Combine:** merge (Algorithm 5) the two sorted subarrays, which takes $\Theta(n)$ time, $C(n) = \Theta(n)$

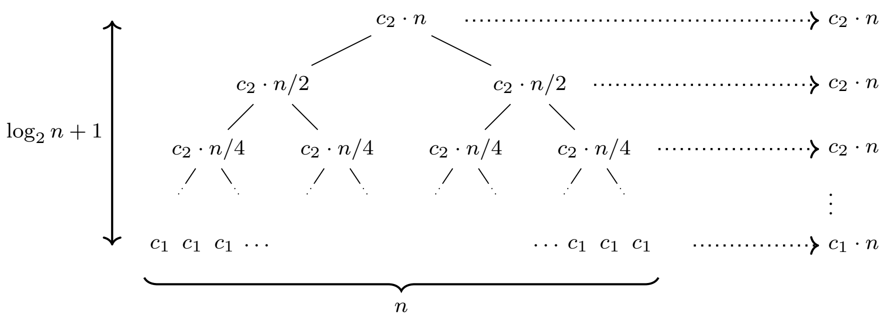
Using the so called 'master theorem':

$$T(n) = 2T(n/2) + \Theta(n) \xRightarrow{\text{master theorem}} T(n) = \Theta(n \log_2 n)$$

Intuitively we can also understand why that is the case without the master theorem. Assume for simplicity that n is an exact power of 2 and that the implicit base case is $n = 1$:

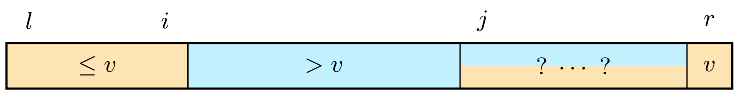
$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{if } n > 1 \end{cases}$$

where $c_1 > 0$ represents the time to solve the base case ($n = 1$) and $c_2 > 0$ is the time per element of the divide and combine steps.



$$\begin{aligned}
T(n) &= 2T(n/2) + c_2 n &&= 2(2T((n/2)/2) + c_2(n/2)) + c_2 n \\
&= 2^2 T(n/2^2) + 2c_2 n &&= 2^2(2T((n/2^2)/2) + c_2(n/2^2)) + 2c_2 n \\
&= 2^3 T(n/2^3) + 3c_2 n &&= 2^3(2T((n/2^3)/2) + c_2(n/2^3)) + 3c_2 n \\
&\vdots \\
&= 2^{\log_2(n)} T(n/2^{\log_2(n)}) + \log_2(n) c_2 n \\
&= nT(1) + \log_2(n) c_2 n \\
&= c_1 n + c_2 n \log_2 n \\
&= \Theta(n \log n)
\end{aligned}$$

3.3 Quick Sort (with Lomuto Partitioning)



Lomuto partitioning (Algorithm 7) divides an array A into two subarrays $A[l : q-1]$ and $A[q+1 : r]$ such that all elements in $A[l : q-1]$ are less than or equal to $A[q]$ and all elements in $A[q+1 : r]$ are greater than $A[q]$.

Algorithm 6 Quick Sort

```
function QuickSort(A, l, r)
    if l < r then
        q = Partition(A, l, r)
        QuickSort(A, l, q − 1)
        QuickSort(A, q + 1, r)
```

Algorithm 7 Lomuto Partitioning

```
function Partition(A, l, r)
    v = A[r]                    ▷ pick last element as pivot
    i = l − 1                   ▷ highest index into the less-than-or-equal–partition
    for j = l, ..., r do
        if A[j] ≤ v then
            i = i + 1
            swap A[i] and A[j]
    return i                    ▷ index of pivot
```

3.4 Heap Sort

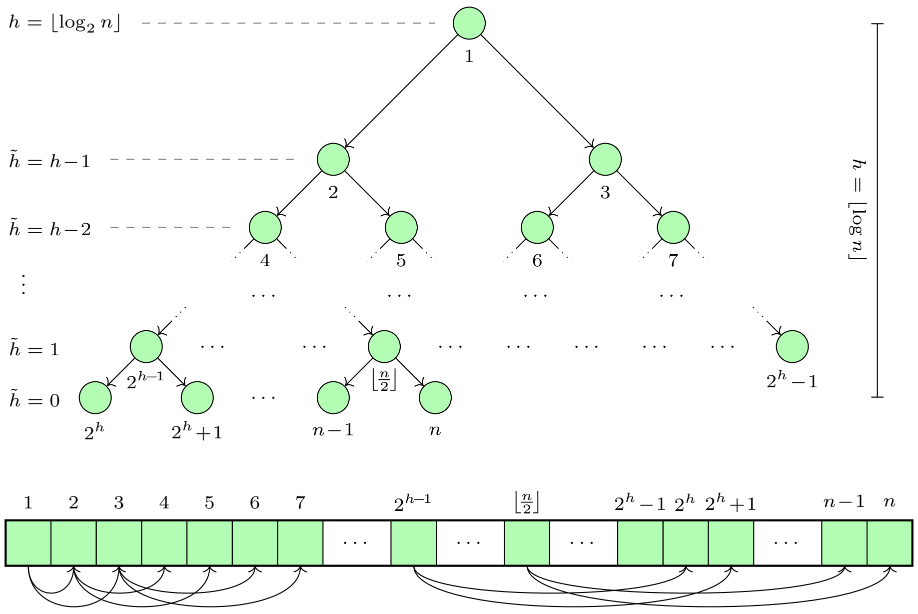
Definition 2 (Heap). A *binary heap* is a nearly-complete binary tree stored in an array $A[1 : n]$ satisfying the *max-heap property*:

$$\forall i > 1 : \quad A[\text{Parent}(i)] \geq A[i]$$

The relationship between the indices of a binary heap is as follows:

$$\text{Parent}(i) = \lfloor i/2 \rfloor \qquad \text{Left}(i) = 2i \qquad \text{Right}(i) = 2i + 1$$

Furthermore, an n-element heap has height $h = \lfloor \log_2 n \rfloor$ and at most $\lceil n/2^{\tilde{h}+1} \rceil$ nodes at any given height $\tilde{h}$, where $\tilde{h}$ is defined as the longest path from the current node to a leaf node, measured in number of edges.



Algorithm 8 Max-Heapify

```
function MaxHeapify(A, i)
    l ← Left(i)
    r ← Right(i)
    m ← i                       ▷ index of largest element among {A[i], A[l], A[r]}
    if l ≤ A.heap-size ∧ A[l] > A[m] then
        m ← l
    if r ≤ A.heap-size ∧ A[r] > A[m] then
        m ← r
    if m ≠ i then
        swap A[i] and A[m]
        MaxHeapify(A, m)
```

Algorithm 9 Build-Max-Heap

```
function BuildMaxHeap(A)
    A.heap-size ← |A|
    for i = ⌊|A|/2⌋, ..., 1 do      ▷ elements after ⌊|A|/2⌋ are leaves
        MaxHeapify(A, i)
```

Algorithm 10 Heap Sort

```
function HeapSort(A)
    BuildMaxHeap(A)
    for i = |A|, ..., 2 do
        swap A[1] and A[i]
        A.heap-size ← A.heap-size − 1
        MaxHeapify(A, 1)
```

Algorithm 10 sorts an array A in place by first building a max-heap from the input array and then repeatedly extracting the maximum element (the root of the heap) and placing it at the end of the array. The complexities of Algorithms 8, 9, 10 are:

$$T_{\text{MaxHeapify}}(n) = \Theta(\log n) \qquad T_{\text{BuildMaxHeap}}(n) = \Theta(n)$$

$$T_{\text{HeapSort}}(n) = \Theta(n \log n)$$

The complexity of Algorithm 8 is determined by the height $\tilde{h}$ of the node to be heapified, which is given by $\tilde{h} = \lfloor \log i \rfloor$ for a node at index i.

Analyzing the complexity of Algorithm 9 is more involved. A simple upper bound on the running time is $O(n \lg n)$, since each call to MaxHeapify costs $O(\log n)$ and BuildMaxHeap makes $O(n)$ such calls. This upper bound is correct but not asymptotically tight.

We can derive a tighter bound by recalling that the time for MaxHeapify to run at a node i depends on the height $\tilde{h}$ of that node in the tree, and that the height of most nodes is small.

Calling MaxHeapify on a node of height $\tilde{h}$ costs $c\,\tilde{h}$, and there are at most $\lceil n/2^{\tilde{h}+1} \rceil$ nodes at that height. We can sum over all heights, starting from the leaves (with height 0) up to the root (with height $\lfloor \log n \rfloor$);

$$T(n) = \sum_{\tilde{h}=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{\tilde{h}+1}} \right\rceil c\,\tilde{h} \leq c\,n \sum_{\tilde{h}=0}^{\lfloor \log n \rfloor} \frac{n}{2^{\tilde{h}}} \tilde{h} \leq c\,n \sum_{\tilde{h}=0}^{\infty} \frac{\tilde{h}}{2^{\tilde{h}}} \tag{3}$$

Recall the geometric series:

$$\sum_{\tilde{h}=0}^{\infty} x^{\tilde{h}} = \frac{1}{1-x}, \quad |x| < 1$$

Differentiating both sides with respect to x gives:

$$\frac{\mathrm{d}}{\mathrm{d}x} \left(\sum_{\tilde{h}=0}^{\infty} x^{\tilde{h}} \right) = \frac{\mathrm{d}}{\mathrm{d}x} \left(\frac{1}{1-x} \right) \implies \sum_{\tilde{h}=0}^{\infty} \tilde{h}\, x^{\tilde{h}-1} = \frac{1}{(1-x)^2}$$

Multiply through by x to shift the exponent back:

$$\sum_{\tilde{h}=0}^{\infty} \tilde{h}\, x^{\tilde{h}} = x \sum_{\tilde{h}=0}^{\infty} \tilde{h}\, x^{\tilde{h}-1} = \frac{x}{(1-x)^2}, \quad |x| < 1, \tag{4}$$

Setting $x = \frac{1}{2}$ in (4) and substituting into (3) gives

$$T(n) \le cn \sum_{\tilde{h}=0}^{\infty} \tilde{h} \left(\frac{1}{2}\right)^{\tilde{h}} = cn \cdot \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = O(n) \tag{5}$$

Thus BuildMaxHeap builds a max-heap from an array in $O(n)$ time.

For Algorithm 10, we have a similar situation, but different from BuildMaxHeap, where the majority of the calls to MaxHeapify are done on nodes at the bottom of the heap (where the height is small), in HeapSort, the calls to MaxHeapify are always done on the root of the heap, and the height is large for the majority of the calls.

After BuildMaxHeap has finished (line 2), the array A is a valid max-heap of size n. The HeapSort loop (lines 3-6) then performs exactly $n-1$ iterations. At the k^{th} iteration the heap contains $n-k+1$ elements, so the call to MaxHeapify costs $\Theta(\log(n-k+1))$. The total time spent in the loop is therefore

$$\sum_{k=1}^{n-1} \Theta\left(\log(n-k+1)\right) \;=\; \Theta\left(\sum_{l=1}^{n-1} \log l\right) \;=\; \Theta\left(\log(n!)\right) \overset{\text{Ex. 3}}{=} \Theta(n \log n)$$

where the change of index $l = n-k+1$ rewrites the sum in increasing order.

3.5 k-Smallest Element

Algorithm 11 is a randomized "divide-and-conquer" method for finding the k-th smallest element in an unsorted array in expected $O(n)$ time. At each step it chooses a pivot uniformly at random, partitions the input into three subsets – those less than, equal to, and greater than the pivot – and then recurses only on the subset that must contain the desired element.

Algorithm 11 Quick Select

```
function QuickSelect(A, k)                     ▷ A is an unordered multiset ('bag') of elements
    v ← A[randint(1, |A|)]                                              ▷ pick a random pivot
    A_L, A_M, A_R ← {}_m                                                 ▷ three empty multisets
    for all a ∈ A do
        if a < v then
            add a to A_L
        else if a = v then
            add a to A_M
        else
            add a to A_R
    if k ≤ |A_L| then
        return QuickSelect(A_L, k)
    else if k ≤ |A_L| + |A_M| then
        return v
    else
        return QuickSelect(A_R, k − (|A_L| + |A_M|))
```

Partitioning around the pivot takes $\Theta(n)$ time, and since the pivot is random the expected size of the recursive call is at most a constant fraction of n, yielding an overall expected running time of $O(n)$.

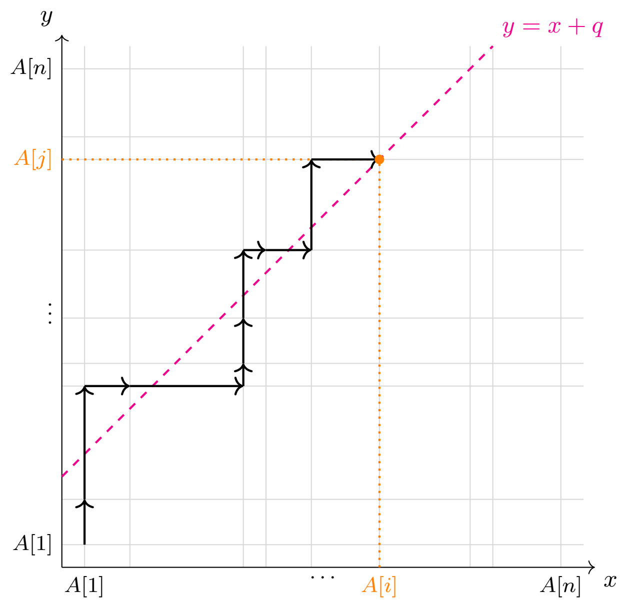
3.6 Overview of Sorting Algorithms

Algorithm	Time Complexity *worst*	*average*	*best*	in place?
InsertionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	✔
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✔
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✘
QuickSort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✔
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✔

3.7 Application

When an array is sorted, many operations and queries (e.g. the one in Example 5) can be performed much more efficiently than in an unsorted array.

Example 5 (Caterpillar Method)**.** Checking if there a sorted array contains two elements $A[i]$ and $A[j]$ such that $A[i] + q = A[j]$, can be done in linear running time using the method (sometimes also called *two-pointer* or *sliding-window* method).



Algorithm 12 Checking for two elements with difference q

```
1:  function Caterpillar(A, q)
2:      l, r ← 1
3:      while r ≤ n do
4:          if A[r] < A[l] + q then
5:              r ← r + 1
6:          else if A[r] > A[l] + q then
7:              l ← l + 1
8:          else
9:              return true
10:     return false
```

Algorithm 12 maintains two indices l and r, both starting at the left end of A (Line 2). At each step, it compares $A[r]$ and $A[l]$ and checks if the difference $A[r] - A[l]$ is less than, greater than, or equal to q:

- If $A[r] - A[l] < q$, move r one step to the right (to increase the difference).
- If $A[r] - A[l] > q$, move l one step to the right (to decrease the difference).
- If $A[r] - A[l] = q$, we found a valid pair and stop. ◀

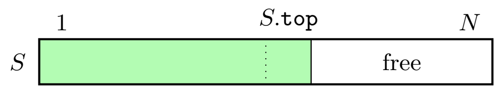
4 Data Structures

Operations on a dynamic set can be grouped into two categories, *queries*, which return information about the set, and *modifying operations*, which change the set:

- Search(S, k)
  query that, given a set S and a key value k, returns a pointer x to an element in S such that x.key $= k$, or nil if no such element belongs to S
- Insert(S, x)
  modifying operation that adds the element pointed to by x to the set S (we usually assume that any attributes in element x needed by the set implementation have already been initialized)
- Delete(S, x)
  modifying operation that, given a pointer x to an element in the set S, removes x from S (note that this operation takes a pointer to an element x, not a key value!)
- Minimum(S) and Maximum(S)
  queries on a totally ordered set S that return a pointer to the element of S with the smallest (for Minimum) or largest (for Maximum) key
- Successor(S, x)
  query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or nil if x is the maximum element
- Predecessor(S, x)
  query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or nil if x is the minimum element

4.1 Stacks

Definition 3 (Stack)**.** A *stack* is a LIFO container that supports constant-time insertion and deletion at one end. ✍



Interface (Algorithm 13):

- StackEmpty(S): returns `true` iff the stack S contains no elements.
- Push(S, x): places element x on the top of the stack S.
- Pop(S): removes and returns the top element of the stack S.

The stack has attributes S.`top`, indexing the most recently inserted element, and S.`length`, equaling the size N of the array.

Algorithm 13 Stack Operations (array-based)

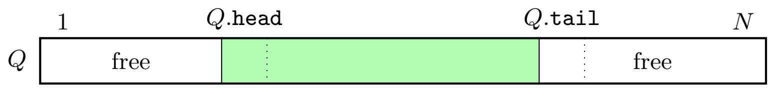
function StackEmpty(S)
  return $(S.\mathtt{top} = 0)$
function Push(S, x)
  if $S.\mathtt{top} = S.\mathtt{length}$ **then**
    error "overflow"
  $S.\mathtt{top} \leftarrow S.\mathtt{top} + 1$
  $S[S.\mathtt{top}] \leftarrow x$
function Pop(S)
  if StackEmpty(S) **then**
    error "underflow"
  $S.\mathtt{top} \leftarrow S.\mathtt{top} - 1$
  return $S[S.\mathtt{top} + 1]$

4.2 Queues

Definition 4 (Queue). A *queue* is a FIFO container with constant-time insertion at the tail and deletion at the head.

Interface (Algorithms 14 and 15):
- Enqueue(Q, x): insert x at the tail of Q.
- Dequeue(Q): remove and return the head element of Q.

The classic fixed-length *circular-array* implementation keeps two indices $Q.\mathtt{head}$ and $Q.\mathtt{tail}$ ($Q.\mathtt{head}$ points to the first element, $Q.\mathtt{tail}$ to the first free slot).



Algorithm 14 Enqueue (circular array)

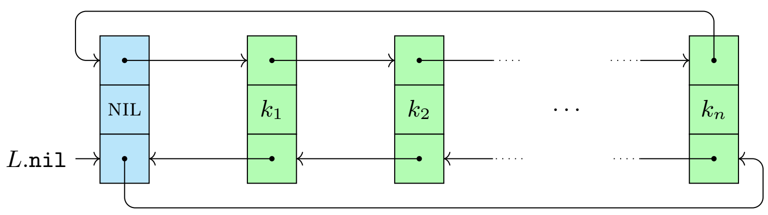
function Enqueue(Q, x)
  if $Q.\mathtt{queue\text{-}full}$ **then**
    error "overflow"
  $Q[Q.\mathtt{tail}] \leftarrow x$
  $Q.\mathtt{tail} \leftarrow (Q.\mathtt{tail} \bmod Q.\mathtt{length}) + 1$
  $Q.\mathtt{queue\text{-}empty} \leftarrow \mathtt{false}$
  if $Q.\mathtt{tail} = Q.\mathtt{head}$ **then**
    $Q.\mathtt{queue\text{-}full} \leftarrow \mathtt{true}$

Algorithm 15 Dequeue (circular array)

function Dequeue(Q)
  if $Q.\mathtt{queue\text{-}empty}$ **then**
    error "underflow"
  $x \leftarrow Q[Q.\mathtt{head}]$
  $Q.\mathtt{head} \leftarrow (Q.\mathtt{head} \bmod Q.\mathtt{length}) + 1$
  $Q.\mathtt{queue\text{-}full} \leftarrow \mathtt{false}$
  if $Q.\mathtt{tail} = Q.\mathtt{head}$ **then**
    $Q.\mathtt{queue\text{-}empty} \leftarrow \mathtt{true}$
  return x

4.3 Linked Lists

Definition 5 (Doubly-Linked List). A *doubly-linked list* L consists of nodes x that store a key $x.\mathtt{key}$ and two links $x.\mathtt{prev}$, $x.\mathtt{next}$. A special sentinel node $L.\mathtt{nil}$ simplifies boundary cases because the list is empty iff $L.\mathtt{nil.next} = L.\mathtt{nil}$.



Algorithm 16 Doubly-Linked List Operations (with sentinel)

function ListInit(L)
  $L.\mathtt{nil.prev} \leftarrow L.\mathtt{nil}$
  $L.\mathtt{nil.next} \leftarrow L.\mathtt{nil}$
function ListInsert(L, x) ▷ insert at front
  $x.\mathtt{next} \leftarrow L.\mathtt{nil.next}$
  $L.\mathtt{nil.next.prev} \leftarrow x$
  $L.\mathtt{nil.next} \leftarrow x$
  $x.\mathtt{prev} \leftarrow L.\mathtt{nil}$
function ListDelete(x)
  $x.\mathtt{prev.next} \leftarrow x.\mathtt{next}$
  $x.\mathtt{next.prev} \leftarrow x.\mathtt{prev}$
function ListSearch(L, k)
  $x \leftarrow L.\mathtt{nil.next}$
  while $x \neq L.\mathtt{nil} \land x.\mathtt{key} \neq k$ **do**
    $x \leftarrow x.\mathtt{next}$
  return x

Algorithm 16 shows the basic operations on a doubly-linked list. ListInsert and ListDelete take $O(1)$ time, whereas ListSearch takes $\Theta(n)$ time in the worst case, with n the current length of L.

4.4 Dictionaries

Definition 6 (Dictionary). A *dictionary* is an abstract data structure that represents a set of elements (or keys). It is a *dynamic set* that supports the following operations:
- Insert: insert element into the set
- Delete: delete element from the set
- Search: test memebership of an element in the set

4.5 Direct-Address Tables

Definition 7 (Direct-Address Table). A *direct-address table* implements a dictionary. Suppose the key universe is the set $U = \{1, \dots, M\}$. A *direct-address table* is an array $T[1 : M]$ where slot $T[k]$ stores a Boolean indicating membership of key k in the represented set.

Algorithm 17 Direct-Address Table Operations

function DirectAddressInsert(T, k)
  $T[k] \leftarrow \mathtt{true}$
function DirectAddressDelete(T, k)
  $T[k] \leftarrow \mathtt{false}$
function DirectAddressSearch(T, k)
  return $T[k]$

All direct-address table operations (Algorithm 17) cost $O(1)$ time, but the table occupies $\Theta(|U|)$ space, which is prohibitive when the universe is large and the actual set is sparse. i.e., direct-address tables usually waste a lot of space.

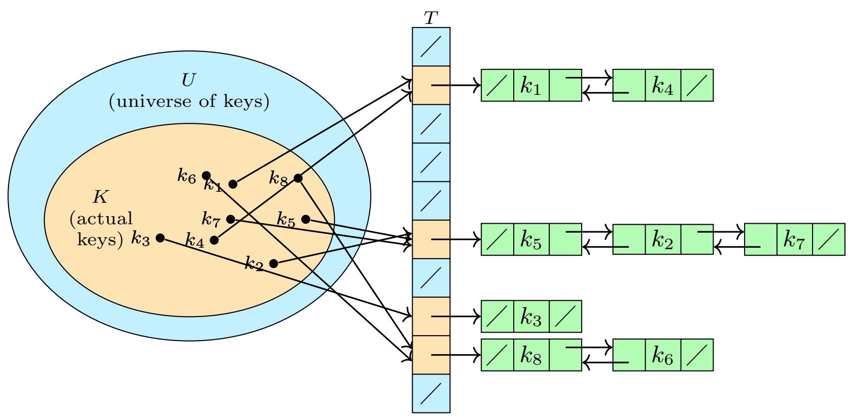
4.6 Hash Tables

To reduce the space overhead we use a smaller table T with $|T| \ll |U|$ and map each key $k \in U$ to a position in T using a *hash function* $h : U \to \{1, \dots, |T|\}$.

Definition 8 (Load Factor). For a table of size $|T|$ that currently stores n keys, the *load factor* is $\alpha = \frac{n}{|T|}$.

4.6.1 Chaining

Each slot $T[i]$ stores a linked list of keys that hash to i.



Algorithm 18 Chained Hash Operations

function ChainedHashInsert(T, k)
  ListInsert($T[h(k)], k$)
function ChainedHashSearch(T, k)
  return ListSearch($T[h(k)], k$)
function ChainedHashDelete(T, k)
  $x \leftarrow$ ChainedHashSearch(T, k)
  if $x \neq$ NIL **then**
    ListDelete(x)

We assume *uniform hashing* i.e.

$$\mathrm{P}[h(k) = i] = \frac{1}{|T|} \quad \forall i \in \{1, \dots, |T|\}$$

So, given n distinct keys, the expected length n_i of the linked list at position i is

$$\mathrm{E}[n_i] = \frac{n}{|T|} = \alpha$$

If we further assume $h(k)$ can be computed in $O(1)$, the expected running time of ChainedHashSearch is

$$\boxed{\Theta(1 + \alpha)}$$

4.6.2 Open Addressing

Instead of using linked lists, keys are stored directly in the array. On a collision we *probe* other slots in T using a permutation $h(k, 1), \dots, h(k, |T|)$. So $h(k, i)$ is a function of both k and i, where i is the probe number.

$h(k, \cdot)$ must be a *permutation* of $\{1, \dots, |T|\}$, i.e., $h(k, 1), \dots, h(k, |T|)$ must cover all slots in T exactly once.

We assume *independent uniform permutation hashing*: the probe sequence of each key is equally likely to be any of the $|T|!$ permutations of $\{1, \dots, |T|\}$. Independent uniform permutation hashing generalizes the notion of independent uniform hashing introduced earlier to a hash function that produces not just a single slot number, but a whole probe sequence. True independent uniform permutation hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, Example 6) are used.

Neither double hashing nor its special case, linear probing, meets the assumption of independent uniform permutation hashing. Double hashing cannot generate more than $|T|^2$ different probe sequences. Nonetheless, double hashing has a large number of possible probe sequences and seems to give good results. Linear probing is even more restricted, capable of generating only $|T|$ different probe sequences.

Example 6 (Double Hashing). Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. Double hashing uses a hash function of the form

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod |T| \tag{6}$$

where h_1 and h_2 are two different hash functions. The second hash function $h_2(k)$ must be *relatively prime* to $|T|$ (i.e., $\gcd(h_2(k), |T|) = 1$) to ensure that all slots in T are probed. A convenient way to achieve this is to let $|T|$ be an exact power of 2 and to design $h_2(k)$ so that it always produces an odd number. Another way is to let $|T|$ be prime and to design $h_2(k)$ so that it always returns a positive integer less than $|T|$. In (6) we can also set $h_2(k) = 1$ for all k, in which case we get *linear probing*, a special case of double hashing. ◂

Algorithm 19 Open-Address Hash Insert (generic probing)

```
function HashInsert(T, k)
    for i = 1, ..., |T| do
        j ← h(k, i)
        if T[j] = NIL then
            T[j] ← k
            return j
    error "overflow"
```

To analyze the time complexity of Algorithm 19, we assume independent uniform permutation hashing for the hash function. We also assume that at least one slot is empty, i.e., $\alpha < 1$. Because deleting from an open-address hash table does not really free up a slot, we assume as well that no deletions occur.

If we denote by X the number of probes performed until an empty slot is found, then on each probe we hit an occupied slot with probability $\alpha = n/|T|$ and an empty slot with probability $1 - \alpha$. Hence

$$\mathrm{P}[X = i] = \alpha^{i-1}(1-\alpha), \quad i = 1, 2, \ldots$$

so that X is geometrically distributed with success probability $1 - \alpha$. Hence

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} i\alpha^{i-1}(1-\alpha) = \frac{1}{1-\alpha}$$

Thus an insertion (or an unsuccessful search) requires on average $\frac{1}{1-\alpha}$ probes; this grows rapidly as the load factor α approaches 1.

4.7 Binary Search Trees

Definition 9 (Binary Search Tree). A *binary search tree* implements a dynamic set. It stores keys from a totally ordered domain in nodes linked by *left* and *right* child pointers such that for every node x

$$y \in \text{left-subtree}(x) \Leftrightarrow y.\texttt{key} \le x.\texttt{key}, \quad z \in \text{right-subtree}(x) \Leftrightarrow z.\texttt{key} \ge x.\texttt{key}$$

✍

Example 7 (Lower Bound). Let t be the root of a binary search tree that represents a set S of numbers. The size of the tree is $|S| = n$. The height of the tree is h. Algorithm 20 returns the node containing the least element $y \in S$ such that $x \le y$, or NIL if no such element exists.

Algorithm 20 Lower Bound for Binary Search Tree

```
function LowerBound(t, x)
    if t = NIL then                          ▷ base case
        return NIL
    if t.key < x then
        return LowerBound(t.right, x)
    else
        y ← LowerBound(t.left, x)
        if y ≠ NIL then
            return y
        else
            return t
```

The complexity is $O(h)$, where h is the height of the tree. ◂

4.7.1 Traversals

Algorithm 21 Inorder Tree Walk (recursive)

```
1: function TreeWalk(x)
2:     if x ≠ NIL then
3:         TreeWalk(x.left)
4:         print x.key
5:         TreeWalk(x.right)
```

Algorithm 21 shows the InOrderTreeWalk algorithm. Three other variants can be obtained from Algorithm 21 by swapping the order of the recursive calls and the print statement (lines 3, 4, and 5). For PreOrderTreeWalk, we swap lines 3 and 4. For PostOrderTreeWalk, we swap lines 4 and 5. For ReverseOrderTreeWalk, we swap lines 3 and 5.

The general recurrence is

$$T(n) = T(n_L) + T(n - n_L - 1) + \Theta(1)$$

and all four variants run in $\Theta(n)$ time. This can be proven using the *substitution method*. Can we do better? No, because the length of the output is $\Theta(n)$.

4.7.2 Basic Queries

Algorithm 22 Searching a Binary Search Tree for a Key

```
function TreeSearch(x, k)
    if x = NIL ∨ k = x.key then
        return x
    if k < x.key then
        return TreeSearch(x.left, k)
    else
        return TreeSearch(x.right, k)
```

Algorithm 23 Minimum of a Binary Search Tree

```
function TreeMinimum(x)
    while x.left ≠ NIL do
        x ← x.left
    return x
```

To find the maximum, replace all occurences of `left` with `right` in Algorithm 23.

Definition 10 (Successor). The *successor* of a node x is the minimum of the right subtree of x if it exists, otherwise it is the lowest ancestor a of x such that x falls in the left subtree of a. ✍

Algorithm 24 Successor of a Node in a Binary Search Tree

```
function TreeSuccessor(x)
    if x.right ≠ NIL then
        return TreeMinimum(x.right)          ▷ leftmost node in right subtree
    while x.parent ≠ NIL ∧ x.parent.right = x do
        x ← x.parent
    return x.parent          ▷ lowest ancestor of x whose left child is an ancestor of x
```

To find the predecessor, replace all occurences of `right` with `left` and use TreeMaximum instead of TreeMinimum in Algorithm 24.

4.7.3 Updates

Algorithm 25 Inserting a Node into a Binary Search Tree

```
function TreeInsert(T, z)
    y, x ← NIL, T.root
    while x ≠ NIL do
        y ← x
        if z.key < x.key then
            x ← x.left
        else
            x ← x.right
    z.parent ← y
    if y = NIL then
        T.root ← z
    else if z.key < y.key then
        y.left ← z
    else
        y.right ← z
```

Deletion distinguishes three cases:

- z is a leaf node, i.e. it has no children:
  - simply remove the node z
- z has one child:
  - remove z
  - connect its child to its parent
- z has two children:
  - find the successor y of z (since z has two children, y is guaranteed to be the minimum of the right subtree of z and thus have at most one child)
  - copy the key of y into z
  - delete y, which is a leaf or has one right child, i.e. connect the child of y to the parent of y

Algorithm 26 Deleting a Node from a Binary Search Tree

```
function TreeDelete(T, z)
    ▷ z has two children: find successor, copy its key, then delete successor ◁
    if z.left ≠ NIL ∧ z.right ≠ NIL then
        s ← TreeMinimum(z.right)
        z.key ← s.key          ▷ replace the key in z with the one from the successor
        return TreeDelete(T, s)
    ▷ z has at most one child: pick it (could be NIL) ◁
    if z.left ≠ NIL then
        c ← z.left
    else
        c ← z.right
    if c ≠ NIL then                        ▷ if child exists, update its parent pointer
        c.parent ← z.parent
    if z.parent = NIL then                 ▷ if z was the root, make child the new root
        T.root ← c
    else                         ▷ otherwise, bypass z by connecting its child to its parent
        if z = z.parent.left then
            z.parent.left ← c
        else
            z.parent.right ← c
    return T
```

Insertion, search and deletion operations have complexity $\Theta(h)$ where h is the height of the tree. In the average case, the height is $O(\log n)$ (i.e. with a random insertion order). In some particular cases, the height can be $O(n)$ (i.e. with ordered sequence). The problem is that the 'worst case' is not that uncommon. One way to avoid this is to instead of inserting $A = [a_1, a_2, a_3, \ldots, a_n]$ in order, insert a random permutation of A. The problem is that A is usually not known in advance. It is the application that calls the insertion procedure. But we can also obtain a random permutation of A by using a randomized insertion algorithm (see 4.7.4).

4.7.4 Randomized Insertion

In order to avoid the linear-height worst case one can insert into a tree using Algorithm 27. The idea behind the function TreeRandomizedInsert is as follows. We insert a new node z into the tree t as the new root of t with probability $1/(t.\texttt{size}+1)$. If z is not inserted as the new root, we recursively insert it into the appropriate subtree of t. The additional attribute $t.\texttt{size}$ represents the number of nodes in the subtree rooted at t.

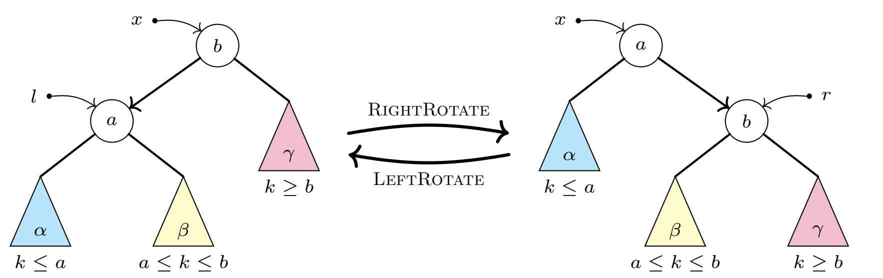
Algorithm 27 Randomized Insertion into a Binary Search Tree

```
function TreeRandomizedInsert(t, z)
    if t = NIL then
        return z
    r ← RANDINT(1, t.size + 1)
    if r = 1 then      ▷ P(r = 1) = P(z is inserted as the new root of t) = 1/(t.size+1)
        z.size ← t.size + 1
        return TreeRootInsert(t, z)                                      ▷ see 4.7.6
    if z.key < t.key then
        t.left ← TreeRandomizedInsert(t.left, z)
    else
        t.right ← TreeRandomizedInsert(t.right, z)
    t.size ← t.size + 1
    return t
```

With Algorithm 27 every insertion order is equally likely; the expected height of the tree is $O(\log n)$ and all operations run in expected $O(\log n)$ time.

4.7.5 Rotations

Definition 11 (Rotation). A *rotation* is a local restructuring of a BST that exchanges the relative position of a node x and one of its children while preserving the in-order sequence of keys. Rotations do *not* change the in-order ordering of the keys and they are the basic tool used by self-balancing trees. ✍



Algorithm 28 Rotations in a Binary Search Tree

```
function RightRotate(x)
    l ← x.left
    x.left ← l.right
    l.right ← x
    return l
function LeftRotate(x)
    r ← x.right
    x.right ← r.left
    r.left ← x
    return r
```

Rotations are very useful operations. For example, we can use it to perform

4.7.6 Root Insertion

Algorithm 29 Root Insertion into a Binary Search Tree

```
function TreeRootInsert(x, z)
    if x = NIL then
        return z
    if z.key < x.key then
        x.left ← TreeRootInsert(x.left, z)
        return RightRotate(x)
    else
        x.right ← TreeRootInsert(x.right, z)
        return LeftRotate(x)
```

General strategies to deal with complexity in the worst case:

- *Randomization*: can make the scenario $h = n$ highly unlikely
- *Amortized Maintenance*: relatively expensive but 'amortized' operations
- *Self-Balancing*: e.g. Red-Black trees (see 4.8)

4.8 Red-Black Trees

A *red-black tree* is a binary search tree with one extra bit of storage per node: its `color`, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Definition 12 (Red-Black Tree). A red-black tree is a binary search tree that satisfies the following *red-black properties*:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. ✍

Definition 13 (Black-Height). The number of black nodes on any simple path from, but <u>not</u> including, a node x down to, and including, a leaf is called the *black-height* of the node x:

$$\mathrm{bh}(x) \coloneqq \#\,\text{black nodes on path from (excluding) } x \text{ to (including) leafs}$$

By property 5, the notion of black-height is well defined, since all descending paths have the same number of black nodes. ✍

Lemma 2 (Height of a Red-Black Tree). The height $h(x)$ of a red-black tree with $n = \mathrm{size}(x)$ internal nodes is at most $2\log(n+1)$. ◁

Proof. First, we prove (by induction) that the subtree rooted at any node x contains at least

$$2^{\mathrm{bh}(x)} - 1 \tag{7}$$

internal nodes.

(i) base case: x is a leaf, so $\mathrm{size}(x) = 0$ and $\mathrm{bh}(x) = 0$, fulfilling (7) ✔

(ii) induction step: consider x, y_1, y_2 such that $x = y_1.\texttt{parent} = y_2.\texttt{parent}$

$$\mathrm{size}(x) = \mathrm{size}(y_1) + \mathrm{size}(y_2) + 1 \geq (2^{\mathrm{bh}(y_1)} - 1) + (2^{\mathrm{bh}(y_2)} - 1) + 1$$

By rule 5, both children must have the same black-height. Let $\mathrm{bh}(y) = \mathrm{bh}(y_1) = \mathrm{bh}(y_2)$. Then

$$\mathrm{size}(x) \geq 2 \cdot (2^{\mathrm{bh}(y)} - 1) + 1 = 2^{\mathrm{bh}(y)+1} - 1$$

The black-height of a child y differs at most by one from the black-height of the parent x, i.e. $\mathrm{bh}(y) \in \{\mathrm{bh}(x), \mathrm{bh}(x) - 1\}$. In both cases, we have $\mathrm{size}(x) \geq 2^{\mathrm{bh}(x)} - 1$, fulfilling (7) ✔

By property 4, the black-height of a node x is at least half the height of the tree, i.e. $\mathrm{bh}(x) \geq \frac{h(x)}{2}$. Therefore,

$$n = \mathrm{size}(x) \geq 2^{\mathrm{bh}(x)} - 1 \geq 2^{\frac{h(x)}{2}} - 1$$

which can be rearranged to

$$h(x) \leq 2\log(n+1)$$ □

5 Graphs

5.1 Representations of Graphs

There are two standard ways to represent a graph: as a collection of adjacency lists or as an adjacency matrix. Because the adjacency-list representation provides a compact way to represent *sparse* graphs (those for which $|E| \ll |V|^2$), it is usually the method of choice. The adjacency-matrix representation might be preferred when the graph is *dense* (i.e., $|E| \approx |V|^2$), or you need to be able to tell quickly wether there is an edge connecting two given vertices.

The space required for the adjacency-list representation is $\Theta(|V| + |E|)$, and finding each edge in the graph also takes $\Theta(|V| + |E|)$ time, since each of the $|V|$ adjacency lists must be examined.

The space required for the adjacency-matrix representation is $\Theta(|V|^2)$, and finding each edge in the graph takes $\Theta(|V|^2)$ time, since the entire adjacency matrix must be examined. For an undirected graph, the adjacency matrix is symmetric, i.e. $\underline{\boldsymbol{A}} = \underline{\boldsymbol{A}}^T$.

The complexity for different operations is summarized in the following table.

Task	Representation	
	Adjacency List	Adjacency Matrix
accessing vertex u	$O(1)$ optimal	$O(1)$ optimal
iteration through V	$\Theta(\|V\|)$ optimal	$\Theta(\|V\|)$ optimal
iteration through E	$\Theta(\|V\|+\|E\|)$ okay (not optimal)	$\Theta(\|V\|^2)$ possibly very bad
checking $(u,v) \in E$	$O(\|V\|)$ bad	$O(1)$ optimal
space complexity	$\Theta(\|V\|+\|E\|)$ optimal	$\Theta(\|V\|^2)$ possibly very bad

For a graph $G = (V, E)$ with source s we define the predecessor subgraph of G as $G_\pi = (V_\pi, E_\pi)$ where

$$V_\pi = \{v \in V \mid v.\pi \neq \text{NIL}\} \cup \{s\}$$

and

$$E_\pi = \{(v.\pi, v) \mid v \in V_\pi \setminus \{s\}\}$$

For depth-first search we define the predecessor subgraph as $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(v.\pi, v) \mid v \in V \wedge v.\pi \neq \text{NIL}\}$$

It is a forest.

5.1.1 Graph Traversals

Algorithm 30 Breadth-First Search

function BFS($G = (V, E)$, s) ▷ s is the source
  for all $u \in V \setminus \{s\}$ **do**
    u.color, u.distance, $u.\pi \leftarrow$ WHITE, ∞, NIL
  s.color, s.distance, $s.\pi \leftarrow$ GRAY, 0, NIL
  $Q \leftarrow \emptyset$ ▷ queue for vertices to visit
  Enqueue(Q, s)
  while $Q \neq \emptyset$ **do**
    $u \leftarrow$ Dequeue(Q)
    for all $v \in \Gamma(u)$ **do**
      if v.color = WHITE **then**
        v.color $\leftarrow$ GRAY
        v.distance $\leftarrow$ u.distance + 1
        $v.\pi \leftarrow u$
        Enqueue(Q, v)
    u.color $\leftarrow$ BLACK

BFS constructs the breadth-first tree G_π and records v.distance $= \delta(s, v)$. Running time: $\Theta(|V| + |E|)$.

Algorithm 31 Depth-First Search

function DFS($G = (V, E)$)
  Initialize v.color $\leftarrow$ WHITE, $v.\pi \leftarrow$ NIL for all $v \in V$
  $T \leftarrow 0$
  for all $v \in V$ **do**
    if v.color = WHITE **then** DFS-Visit(v)
function DFS-Visit(u)
  u.color $\leftarrow$ GRAY
  $T \leftarrow T + 1$
  u.disc $\leftarrow T$
  for all $v \in \Gamma(u)$ **do**
    if v.color = WHITE **then**
      $v.\pi \leftarrow u$
      DFS-Visit(v)
  u.color $\leftarrow$ BLACK
  $T \leftarrow T + 1$
  u.finish $\leftarrow T$

DFS runs in $\Theta(|V| + |E|)$.

5.1.2 Minimum Spanning Trees

For a connected, undirected, weighted graph $G = (V, E, w)$ an MST is a subset of edges that connects all vertices with minimum total weight.

5.1.3 Kruskal

Algorithm 32 Kruskal

function Kruskal($G = (V, E, w)$)
  $T \leftarrow \emptyset$
  MakeSet(v) for each $v \in V$
  for all $(u, v) \in E$ in non-decreasing w-order **do**
    if Find(u) $\neq$ Find(v) **then**
      $T \leftarrow T \cup \{(u, v)\}$
      Union(u, v)
  return T

5.1.4 Prim

Algorithm 33 Prim (binary heap)

function Prim($G = (V, E, w)$, r)
  Initialize v.key $\leftarrow \infty$, $v.\pi \leftarrow$ NIL for all $v \in V$
  r.key $\leftarrow 0$
  $Q \leftarrow V$ ▷ min-priority queue by .key
  while $Q \neq \emptyset$ **do**
    $u \leftarrow$ ExtractMin(Q)
    for all $v \in \Gamma(u)$ **do**
      if $v \in Q$ and $w(u, v) < v$.key **then**
        $v.\pi \leftarrow u$
        v.key $\leftarrow w(u, v)$
        DecreaseKey(Q, v, v.key)
  return $\{(v.\pi, v) \mid v \in V \setminus \{r\}\}$

Both Kruskal and Prim run in $\Theta(|E| \log |V|)$ with binary heaps.

5.2 Single-Source Shortest Paths

5.2.1 Dijkstra (non-negative weights)

Algorithm 34 Dijkstra

function Dijkstra($G = (V, E, w)$, s)
  Initialize v.distance $\leftarrow \infty$, $v.\pi \leftarrow$ NIL for all $v \in V$
  s.distance $\leftarrow 0$
  $Q \leftarrow V$ ▷ min-queue keyed by .distance
  while $Q \neq \emptyset$ **do**
    $u \leftarrow$ ExtractMin(Q)
    for all $(u, v) \in E$ **do**
      if v.distance $>$ u.distance $+ w(u, v)$ **then**
        v.distance $\leftarrow$ u.distance $+ w(u, v)$
        $v.\pi \leftarrow u$
        DecreaseKey(Q, v, v.distance)
  return $(v.\text{distance})_{v \in V}$

Binary heap: $\Theta(|E| \log |V|)$; Fibonacci heap: $\Theta(|E| + |V| \log |V|)$.

5.3 Complexity Overview

Algorithm	Adjacency list	Conditions
BFS, DFS	$\Theta(\|V\|+\|E\|)$	–
Kruskal	$\Theta(\|E\| \log \|V\|)$	sort + DSU
Prim (bin. heap)	$\Theta(\|E\| \log \|V\|)$	–
Dijkstra (bin.)	$\Theta(\|E\| \log \|V\|)$	$w \geq 0$