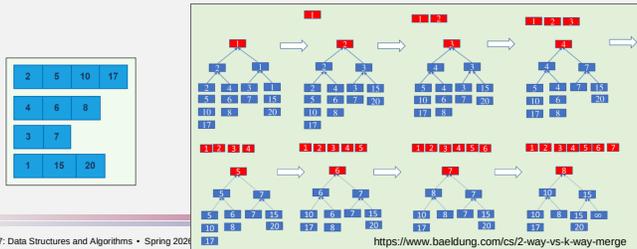


Sorting – Key Algorithms

external sorting | any technique used for sorting data sets that are too big to fit into memory

- multiway mergesort
 - sort chunks that are small enough to fit into memory
 - iterative 2-way merge
 - repeatedly merge pairs of chunks
 - direct k -way merge
 - merge k chunks at once, maintaining a heap or tournament tree of the next element in each chunk to have an efficient way to find the next smallest element



CSPC 327: Data Structures and Algorithms • Spring 2026

<https://www.baeldung.com/cs/2-way-vs-k-way-merge>

Sorting

- best general-purpose sorting algorithm is (randomized) quicksort
 - use a library implementation because there are many details to achieve maximum performance
- heapsort is a good choice if you have to implement a sorting algorithm
 - $O(n \log n)$
 - easy to implement, especially with a priority queue available
 - can be implemented in-place

“Although other algorithms prove slightly faster in practice, you won’t go wrong using heapsort for sorting data that sits in the computer’s main memory.” [Skienna p. 121]

CSPC 327: Data Structures and Algorithms • Spring 2026

200

Sorting

Choosing a sorting algorithm –

ADM 17.1

- how many keys are being sorted?
 - for small n (≤ 100), $O(n^2)$ is fine and simple implementations are preferred
 - for medium and large n , $O(n \log n)$ is needed
 - for very large n ($\geq 100,000,000$), need external sorting
- are there duplicate keys?
 - a *stable* sorting algorithm keeps equal keys in the original order
 - many algorithms are not inherently stable, but can be made so by including the initial position as a secondary key
 - having many equal keys amongst the elements being sorted can lead to performance problems for some algorithms if not explicitly designed for it
- does the data have special properties?
 - is it already partially sorted?
 - are the keys uniformly distributed across the range of values?
 - are the keys long?
 - is the range of keys very small?

11

Sorting in Java – Arrays

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html>

static void	sort(int[] a)	Sorts the specified array into ascending numerical order.
static void	sort(int[] a, int fromIndex, int toIndex)	Sorts the specified range of the array into ascending order.

- plus similar versions for arrays of the other primitive types

sort

```
public static void sort(int[] a)
```

Sorts the specified array into ascending numerical order.

Implementation Note:

The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers $O(n \log(n))$ performance on all data sets, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

a - the array to be sorted

CSPC 327: Data Structures and Algorithms • Spring 2026

202

Sorting in Java – Arrays

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html>

static void	sort(Object[] a)	Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.
static void	sort(Object[] a, int fromIndex, int toIndex)	Sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements.
static <T> void	sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)	Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.
static <T> void	sort(T[] a, Comparator<? super T> c)	Sorts the specified array of objects according to the order induced by the specified comparator.

- *natural ordering* means the elements must implement the Comparable interface

Sorting in Java – Arrays

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than $n \lg(n)$ comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to $n/2$ object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort[®]). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Parameters:

a - the array to be sorted

Throws:

`ClassCastException` - if the array contains elements that are not *mutually comparable* (for example, strings and integers)

`IllegalArgumentException` - (optional) if the natural ordering of the array elements is found to violate the Comparable contract

Sorting in Java – Collections

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html>

static <T extends Comparable<? super T>> void	sort(List<T> list)	Sorts the specified list into ascending order, according to the natural ordering of its elements.
static <T> void	sort(List<T> list, Comparator<? super T> c)	Sorts the specified list according to the order induced by the specified comparator.

- *natural ordering* means the elements must implement the Comparable interface

Implementation Note:

This implementation defers to the `List.sort(Comparator)` method using the specified list and a null comparator.

Sorting in Java – List

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html>

```
default void sort(Comparator<? super E> c)
```

Sorts this list according to the order induced by the specified Comparator. The sort is *stable*: this method must not reorder equal elements.

All elements in this list must be *mutually comparable* using the specified comparator (that is, `c.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

If the specified comparator is null then all elements in this list must implement the Comparable interface and the elements' natural ordering should be used.

This list must be modifiable, but need not be resizable.

Implementation Requirements:

The default implementation obtains an array containing all elements in this list, sorts the array, and iterates over this list resetting each element from the corresponding position in the array. (This avoids the $n^2 \log(n)$ performance that would result from attempting to sort a linked list in place.)

Implementation Note:

This implementation is a stable, adaptive, iterative mergesort that requires far fewer than $n \lg(n)$ comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to $n/2$ object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort[®]). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Comparing in Java

- when there is a natural ordering, the type itself implements `Comparable`

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html>

- when there is not a natural ordering, define a standalone *comparator*
 - separate class implementing `Comparator`

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Comparator.html>

Comparable

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure $\text{signum}(x.\text{compareTo}(y)) == -\text{signum}(y.\text{compareTo}(x))$ for all x and y . (This implies that $x.\text{compareTo}(y)$ must throw an exception if and only if $y.\text{compareTo}(x)$ throws an exception.)

The implementor must also ensure that the relation is transitive: $(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0)$ implies $x.\text{compareTo}(z) > 0$.

Finally, the implementor must ensure that $x.\text{compareTo}(y) == 0$ implies that $\text{signum}(x.\text{compareTo}(z)) == \text{signum}(y.\text{compareTo}(z))$, for all z .

API Note:

It is strongly recommended, but *not* strictly required that $(x.\text{compareTo}(y) == 0) == (x.\text{equals}(y))$. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

Parameters:

`o` - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

`NullPointerException` - if the specified object is null

`ClassCastException` - if the specified object's type prevents it from being compared to this object.

Comparator

```
int compare(T o1,  
           T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

The implementor must ensure that $\text{signum}(\text{compare}(x, y)) == -\text{signum}(\text{compare}(y, x))$ for all x and y . (This implies that $\text{compare}(x, y)$ must throw an exception if and only if $\text{compare}(y, x)$ throws an exception.)

The implementor must also ensure that the relation is transitive: $((\text{compare}(x, y) > 0) \ \&\& \ (\text{compare}(y, z) > 0))$ implies $\text{compare}(x, z) > 0$.

Finally, the implementor must ensure that $\text{compare}(x, y) == 0$ implies that $\text{signum}(\text{compare}(x, z)) == \text{signum}(\text{compare}(y, z))$ for all z .

API Note:

It is generally the case, but *not* strictly required that $(\text{compare}(x, y) == 0) == (x.\text{equals}(y))$. Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

Parameters:

`o1` - the first object to be compared.

`o2` - the second object to be compared.

Returns:

a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Throws:

`NullPointerException` - if an argument is null and this comparator does not permit null arguments

`ClassCastException` - if the arguments' types prevent them from being compared by this comparator.

Searching – Key Points

- searching vs lookup
 - more efficient algorithms are possible when the collection of elements is fixed (no insertions or deletions)
- sequential search
 - can be applied to any traversable collection
- binary search
 - requires a sorted array
 - deceptively tricky to implement correctly – thorough testing requires searching for every key as well as every option for between keys

Searching – Key Points

- how many elements?
 - sequential search is fastest for small collections (≤ 20), even if they are already sorted
 - beyond 100 elements (and especially if multiple searches), better to sort first and use binary search
 - for huge data sets (too big for memory), use B-trees or another data structure
- are some elements accessed more often than others?
 - take advantage – order list accordingly or build optimal binary search tree
- do access frequencies change over time?
 - *self-organizing* structures – *move-to-front heuristic* (lists), *splay trees*
- unbounded range, or element thought to be nearby?
 - *one-sided binary search* – repeatedly double the search interval size, extending one side of the interval, until the upper bound is found (then regular binary search)

Selection – Key Points

- selection – find k th element (commonly median)
 - searching locates an element by value, selection locates an element by rank
- applications
 - order statistics ($k = 1, k = n, k = n/2$ – min, max, median)
 - quantiles – median, quartiles, deciles, percentiles, ...
 - filtering – extract or eliminate top/bottom percentage

Selection – Key Points

- how fast do you need?
 - find k th is $\Theta(1)$ in a sorted array – can be worthwhile to sort if you need multiple k values
 - (randomized) *quick-select* – $\Theta(n)$ expected, $\Theta(n^2)$ worst-case (unlikely)
 - based on quicksort – pick pivot, divide into smaller and bigger groups, but then can determine which group will contain the k th element and recurse only on that
- too much data to fit in memory, or to even contemplate storing?
 - *random sampling* to save only a sample of the data set for processing
 - compute quantiles for chunks of data, then combine to approximate the overall measures

Applications of Hashing

- hashing has (many) applications beyond lookup
 - data integrity and verification
 - checksums
 - data deduplication and managing collections of unique elements
 - pattern matching
 - compare hashes to quickly check if two strings are not equal – only check characters if the hashes match
 - cryptography and data security
 - store hashed passwords instead of plaintext
 - digital signatures
- key properties
 - hash function is deterministic
 - (a good) hash function distributes keys well / few collisions
 - hash function is not reversible

Recap – ADTs

We've considered major categories of ADTs for collections, characterized by the access they provide for their elements, and common ADTs within those categories

- **containers** – based on position, not element value
 - Sequence/List – linear structure with access at any position
 - Stack – insert/remove at the same end (top)
 - Queue – insert/remove at opposite ends (front, back)
- **dictionary** – based on element's key (lookup)
 - Dictionary/Map – find(k), insert(k,v), remove(k)
 - OrderedDictionary – also max/min, successor(k), predecessor(k)
- **priority queue** – ordered, based on element's key
 - PriorityQueue – insert(x), findMin (or max), removeMin (or max)

Recap – Data Structures

We've seen some useful data structures – arrays, linked lists, binary trees, general trees.

We've seen some clever ways to use and adapt basic data structures to achieve efficient implementations of ADTs.

- sorted array/linked list (vs. unsorted)
- circular arrays
- linked list with tail pointers
- array-based implementation of binary trees
- search trees (binary, multiway) – trees with ordering property for elements
- balanced search trees (AVL, 2-4, red-black) – search trees with structural property to maintain log n height
- splay trees – search trees with heuristics to keep height log n
- hashtables – arrays with clever conversion of key to array index
- heaps – trees with ordering + structural properties

Specialized Data Structures

Be aware that there's more out there.

- other implementations
 - dictionaries: b-trees, skip lists
 - priority queues: bounded height PQs, Fibonacci heaps, pairing heaps
- string data structures
 - e.g. suffix trees/arrays for pattern matching
 - e.g. prefix trees
- geometric data structures
 - e.g. BSP, kd-trees for fast searching in space
- graph data structures
- set data structures

A Practical Guide to Data Structures and Algorithms using Java, Goldman and Goldman

The Algorithm Design Manual, Skiena



Choosing Data Structures

“Building algorithms around data structures such as dictionaries and priority queues leads to both clean structure and good performance.” [Skienna, ADM]

- first design the ADT – identify how your collection is accessed and what operations are needed
- then choose an implementation that delivers the necessary performance
- isolate the implementation of the data structure from the rest of the code
 - in Java, this means writing a class to implement the ADT with methods for the ADT operations

Choosing Implementations

Consider the characteristics of your task –

- dictionaries
 - how many items? is the size known in advance?
 - if small, simplicity of implementation is most important
 - if very large, running out of memory is an issue
 - what are the relative frequencies of insert, delete, find operations?
 - static (no modifications after construction) and semi-dynamic structures (insertion but no deletion) can be simpler than fully dynamic
 - is the access pattern for keys uniform and random?
 - in some data structures, non-uniform distributions lead to worst-case performance while others can take advantage of temporal locality
 - do individual operations need to be fast, or just minimize the total amount of work of the whole program?
 - focus on achieving good worst case performance vs amortized or expected performance

Implementation Choices for Dictionaries

- for small data sets, unsorted arrays are simple and have better cache performance than linked lists
- for moderate-to-large data sets, hashtables are likely best
- for very large data sets where there isn't enough room in memory, use B-trees
- *self-organizing lists* (move-to-front heuristic) are often better than sorted or unsorted lists in practice
 - many applications have uneven access frequencies and locality of reference
- sorted arrays OK if not too many insertions or deletions
- the inability to use binary search makes sorted linked lists often not worth it
- for balanced search trees, the best choice is likely the one with the best implementation
- *skip lists* are easier than balanced search trees to implement and analyze

Implementation Choices for Dictionaries

- creating good hashables
 - open addressing has better cache performance, but overall performance decreases quickly with higher load factors
 - with open addressing, N should be 30-50% larger than the maximum number of elements expected at once
 - N should be prime
 - use a good hash function + an efficient implementation

$$H(S) = a^{|S|} + \sum_{i=0}^{|S|-1} a^{|S|-(i+1)} \times \text{char}(s_i) \pmod{m}$$

- gather stats on the distribution of keys to see how well the hash function performs

Choosing Implementations

Consider the characteristics of your task –

- priority queues
 - max size? is it known in advance?
 - preallocating the necessary space saves having to grow a container
 - are the key values limited?
 - what operations are needed?
 - if no insertion after construction, no need for PQ – just sort
 - other operations: searching for or removing arbitrary elements vs only the min/max
 - can priorities of elements already in the PQ be changed?
 - implies needing to retrieve elements by key, not just the min/max ones

Implementation Choices for Priority Queues

- sorted array or list when there aren't any insertions
 - very efficient for identifying and removing the smallest element
- binary heaps when the max number of elements is known
 - fixed array size can be mitigated with dynamic arrays
- *bounded-height priority queues* when there is a small, discrete range of keys
- BSTs when other dictionary operations are needed, or when there is an unbounded key range and the max PQ size isn't known in advance
- *Fibonacci and pairing heaps* improve the efficiency of decrease key operations
 - effective for large computations if implemented and used well

Designing Your Own Data Structures

- know what kinds of structures lead to what kinds of running times
 - can use that knowledge to guide/constrain thinking
 - $O(n \log n)$ or better is typically required in practice for large data sets
- strategies for rolling your own data structures
 - store more information for faster access – as long as it can be kept up-to-date efficiently
 - add additional properties to speed desired operations – as long as they can be maintained efficiently
- knowledge of complexity is useful
 - for NP-complete problems, look for heuristics rather than optimal solutions

Designing Data Structures

“...in practice, it is more important to avoid using a bad data structure than to identify the single best option available.”

[Skiena, ADM]

- ask “do we need to do better?” before “can we do better?”