Throughout the course, when designing algorithms, along with lists, stacks, queues, arrays, trees, and graphs, the following three basic data structures (heaps, binary search trees, hash tables) will also be useful.

Priority queues / heaps. A priority queue is a data type for holding a dynamic collection of items, where each item has a *key* (a value such as a number, from some ordered set). It supports the following operations:

INSERT(x) — insert x into the collection.

$x \leftarrow$ FIND-MIN() — return an object in the collection whose key is minimum.

$x \leftarrow$ EXTRACT-MIN() — return and delete an object from the collection whose key is minimum.

DECREASE-KEY(x, k) — lower object x's key to k.

A *heap* is a particular data structure of this type. It supports all operations in $O(\log n)$ time, where n is the number of keys in the collection.

The standard implementation of a heap is as a rooted, complete, binary tree, where the nodes in the tree are the objects in the collection, and the objects are *heap-ordered*: each object's key is as large as its parent's key. Hence, the object at the root has a minimum key.

Typically the n nodes of the tree (one per object currently in the collection) are stored in the first n entries of an array $A[1..N]$, where $N \geq n$ is an upper bound on the maximum collection size. Array cell $A[1]$ is the root. For each array cell $A[i]$, the left child of $A[i]$ (if it exists) is $A[2i]$, and the right child of $A[i]$ (if it exists) is $A[2i+1]$. It follows that the parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$. So $A[i]$ has depth (i.e., distance to the root) at most $\log_2 i$, and the tree has maximum depth $O(\log n)$.

Next we describe how to implement the various operations. Each operation will move the objects within the array A. As it does so, we maintain with each object x its current location $A[i]$ within A, so that, given x, we can determine its location in constant time. We don't discuss this explicitly below.

FIND-MIN() returns the object at the root. This takes constant time.

DECREASE-KEY(x, k) is implemented as follows. Let $A[i]$ be the location of x. Decrease the key of $A[i]$ to k. This could violate heap order, if $A[i]$'s key is smaller than its parent's key. If that's the case, swap $A[i]$ with its parent within the array. Repeat this as needed (possibly bubbling the new object all the way up to the root) until heap order is restored. The total time is $O(\log n)$, because the tree depth is $O(\log n)$.

INSERT(x) is implemented as follows. First increase n by 1, then put x in cell $A[n]$ with a "fake" key, say, $-\infty$. Then do DECREASE-KEY(x, k) to change x's key to its correct key. The total time is $O(\log n)$.

EXTRACT-MIN() is implemented as follows. Record the object x currently at the root $A[1]$, with minimum key. Swap the root $A[1]$ with the rightmost leaf $A[n]$. Decrease n by 1, effectively removing $A[n]$. Now, heap order may be violated at the root, if $A[1]$'s key is larger than the key of either its left or right child. If that's the case, swap $A[1]$ with whichever child has the smallest key. Now, heap order may be violated at that child. If so, swap that child with whichever of its two children has the smallest key. Repeat recursively, as necessary, following a path down the tree, until heap order is restored. The total time is $O(\log n)$.

It is not hard to use the same ideas to implement the following additional operations in $O(\log n)$ time per operation: DELETE(x) and INCREASE-KEY(x, k). We leave this as an exercise.

See also CLRS Chapter 6 and https://en.wikipedia.org/wiki/Priority_queue. CLRS discusses heaps that support FIND-MAX and EXTRACT-MAX instead of FIND-MIN and EXTRACT-MIN. The implementation of these is symmetric (just reverse the ordering on the keys).

Binary search trees. A *binary search tree* is also a rooted binary tree whose nodes represent a dynamic collection of objects that have keys. But it differs from a heap as follows. It is not stored in an array as a heap is — instead, it is stored in standard binary-tree representation, where each node has pointers to its parent and its two children. Instead of the heap-order property, it maintains *search-tree* order: at each node, the key is as large as each key in its left subtree, and no larger than any keys in its right subtree. Thirdly, it supports a more general set of operations, including the following operations:

SEARCH(k) — return an object with key k, if there is one.

INSERT(x) — add object x to the collection.

MINIMUM() — return an object with minimum key.

SUCCESSOR(x) — return the *successor* of x (x must be in the collection). The successor is the object y (if any) whose key is the smallest key larger than x's key.

DELETE(x) — remove object x from the collection.

Disclaimer. We consider here only the special case when all object keys are distinct, so that, for any given key k, there is at most one object in the collection with that key. We enforce this by defining INSERT(x) so that, if the collection already has an object with y the same key as x, INSERT(x) removes y from the collection and replaces it with x. Allowing duplicate keys is not hard, but the details would obscure the discussion.

Next we sketch how to implement the above operations. Each operation will take time $O(d)$, where d is the maximum depth of any leaf. A plain binary search tree is not guaranteed to be balanced. Indeed, in the worst case its depth can be $\Omega(n)$. So, this is not a particularly good run-time guarantee. (More sophisticated variants of binary trees do guarantee depth $O(\log n)$. We will see some of these later in the course.)

SEARCH(k) starts at the root. It compares k to the root's key. If k equals the roots key, it returns the root. Otherwise, it recursively searches the left or right subtree, depending on whether k is less than or greater than the root's key.

INSERT(x) searches for x's key k, as described above. If it finds an object y with key k, then it replaces y with x. Otherwise, the search ends at an empty subtree that is the (non-existent) child of the last node z on the search path, and INSERT(x) inserts x as the actual child of z.

MINIMUM() returns the lowest node on the left spine of the tree. This is found by starting at the root, and following left children until a node is reached that has no left child.

SUCCESSOR(x) is implemented as follows. If the right subtree of x is non-empty, return the minimum node in the right subtree of x (this is the lowest node on the left spine of the left subtree). Otherwise, return the lowest ancestor y of x such that the key of y is larger than the key of x (that is, such that x is in y's left subtree). If there is no such ancestor, then x has no successor.

DELETE(x) is implemented roughly as follows. If x has no children, it is simply removed from the tree (by changing its parent's pointer to NIL, say). If x has only one child, y, then x is spliced out of the tree, by making y take x's place in x's parent. The remaining case, when x has two children, is somewhat complicated. One way to handle it is as follows. Find x's successor, say y, in the tree. Note that y is the lowest node on the left spine in x's right subtree, so has no left child. Delete y

from its current location, then replace x by y. Note that y has no left child, so its deletion uses one of the two simpler cases.

See also: CLRS Ch. 12 (omit 12.4); https://en.wikipedia.org/wiki/Binary_search_tree

Dictionaries / hash tables. A dictionary data type, which is what a hash table implements, supports a subset of the binary-search-tree operations. Namely, the order-independent operations:

SEARCH(k) — return an object with key k, if there is one.

INSERT(x) — add object x to the collection.

DELETE(x) — remove object x from the collection.

Such a data type is also called an *associative array*.

For now, the main thing to know is that *a hash table (if properly implemented) can support each of these operations in constant time per operation.*

For background, here is a rough sketch of how hash tables are implemented. The basic idea is to maintain an array $A[0..N]$ of size $N = \Theta(n)$, and to store an object x in the cell $A[i]$ such that $i = f(k)$, where k is the key of x, and f is a so-called *hash function*, which maps each possible key k to some array index i.

The main challenge is that two different keys k and $k' \neq k$ may hash to the same index i (so $f(k) = i = f(k')$). There are two main approaches for handling this. With *chaining*, each array cell $A[i]$ stores a list ("chain", or "bucket") containing all objects that have been inserted whose keys hash to i.

With *open addressing*, each array cell $A[i]$ holds at most one object x. We need that the number N of cells in A exceeds n by at least, say, a factor of 2. That is, at most half the cells should be occupied. We require the hash function f to take *two* arguments: the key k and an additional index $j \geq 0$. Then, to INSERT(x), we examine the sequence of cells $f(k,0), f(k,1), f(k,2), \ldots$ and use the first one that is empty. To SEARCH(k), we check cells $f(k,0), f(k,1), f(k,2), \ldots$ stopping as soon as we encounter either an object x has key k, or an empty cell (in which case there is no object with key k). The final operation, DELETE(x), is complicated — simply removing the object from its cell would break the SEARCH operation. Instead, we leave it occupied but flag it as "deleted". Then, when the number of deleted cells reaches some threshold (say one quarter of all cells), we completely rebuild the hash table by inserting all non-deleted objects into a new hash table. This operation is expensive, but infrequent.

The above description leaves many details unspecified. In particular, how to choose a good hash function. We will return to that topic later in the course.

See also CLRS Chapter 11 (omit 11.3, 11.5) and https://en.wikipedia.org/wiki/Hash_table