# Part C: Data Structures

Wayne Goddard, School of Computing, Clemson University, 2019

<table>
<thead>
<tr>
<th>Data Structures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1.1 Basic Collections</td>
<td>1</td>
</tr>
<tr>
<td>C1.2 Stacks and Queues</td>
<td>1</td>
</tr>
<tr>
<td>C1.3 Dictionary</td>
<td>2</td>
</tr>
<tr>
<td>C1.4 Priority Queue</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Skip Lists</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2.1 Levels</td>
<td>3</td>
</tr>
<tr>
<td>C2.2 Implementation of Insertion</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Search Trees</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3.1 Binary Search Trees</td>
<td>5</td>
</tr>
<tr>
<td>C3.2 Insertion and Deletion in BST</td>
<td>5</td>
</tr>
<tr>
<td>C3.3 Rotations</td>
<td>6</td>
</tr>
<tr>
<td>C3.4 AVL Trees</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heaps and Binomial / Fibonacci Heaps</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>C4.1 Heaps</td>
<td>9</td>
</tr>
<tr>
<td>C4.2 Heap Operations</td>
<td>9</td>
</tr>
<tr>
<td>C4.3 Binomial Heaps</td>
<td>11</td>
</tr>
<tr>
<td>C4.4 Decrease-Key and Fibonacci Heaps</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash Tables</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5.1 Buckets and Hash Functions</td>
<td>14</td>
</tr>
<tr>
<td>C5.2 Collision Resolution</td>
<td>14</td>
</tr>
<tr>
<td>C5.3 Rehashing</td>
<td>15</td>
</tr>
<tr>
<td>C5.4 Other Applications of Hashing</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disjoint Set Data Structure</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6.1 Disjoint Forests</td>
<td>17</td>
</tr>
<tr>
<td>C6.2 Pointer Jumping</td>
<td>18</td>
</tr>
</tbody>
</table>
Chapter C1: Data Structures

An ADT or abstract data type defines a way of interacting with data: it specifies only how the ADT can be used and says nothing about the implementation of the structure. An ADT is conceptually more abstract than a Java interface specification or C++ list of class member function prototypes, and should be expressed in some formal language (such as mathematics).

A data structure is a way of storing data that implements certain operations. When choosing a data structure for your ADT, you might consider many issues such as whether the data is static or dynamic, whether the deletion operation is important, whether the data is ordered, etc.

C1.1 Basic Collections

There are three basic collections.

1. The basic collection is often called a bag. It stores objects with no ordering of the objects and no restrictions on them.

2. Another unstructured collection is a set where repeated objects are not permitted: it holds at most one copy of each item. A set is often from a predefined universe.

3. A collection where there is an ordering is often called a list. Specific examples include an array, a vector and a sequence. These have the same idea, but vary as to the methods they provide and the efficiency of those methods.

The Bag ADT might have accessor methods such as size, countOccurrences, and an iterator; modifier methods such as add, remove, and addAll; and also a merge method that combines two bags to produce a third.

C1.2 Stacks and Queues

A linear data structure is one which is ordered. There are two special types with restricted access: a stack and a queue.

A stack is a data structure of ordered items such that items can be inserted and removed only at one end (called the top). It is also called a LIFO structure: last-in, first-out. The standard (and usually only) modification operations are:

- push: add the element to the top of the stack
- pop: remove the top element from the stack and return it
If the stack is empty and one tries to remove an element, this is called *underflow*. Another common operation is called *peek*: this returns a reference to the top element on the stack (leaving the stack unchanged).

A queue is a linear data structure that allows items to be added only to the rear of the queue and removed only from the front of the queue. Queues are *FIFO* structures: first-in first-out. The two standard modification methods are:

- insert the item at the *rear* of the queue
- delete and return the item at the *front* of the queue

**C1.3 Dictionary**

A dictionary is also known as an *associative data structure* or a *map*. The dictionary ADT supports:

- **insert**: Insert new item
- **lookup**: Look up item based on key; return access/boolean

There are several implementations: for example, red-black trees do both operations in $O(\log n)$ time. But one can do better by allowing the dictionary to be unsorted. See hash tables later.

**C1.4 Priority Queue**

The (min)-priority queue ADT supports:

- **insert**: Insert new item
- **extractMin**: Remove and return item with minimum key

Other common methods include **decreaseKey** and **delete**.

Note that a priority queue provides a “natural” way to sort: simply

*insert all elements and then repeatedly extractMin*

**Exercises**

1. [Add Me]
Chapter C2: Skip Lists

Skip lists support a dictionary (and more) for data that has an underlying ordering. For example, they can do the dictionary operations on a sorted set. They were introduced by Pugh.

C2.1 Levels

The idea is to use a sorted linked list. Of course, the problem with a linked list is that finding the data and/or finding the place to insert the new data is $O(n)$. So the solution is to have pointers that traverse the list faster (e.g. every 2nd node), and then have pointers that traverse the list even faster (e.g. every 4th node), and so on.

So we will have levels of linked lists. These get sparser as one goes up the levels. Importantly, each node in a level has a pointer to the corresponding node in the level below. Thus, the find operation starts in the topmost level, and moves down a level each time it finds the bracketing pair.

In an ideal scenario, the first linked list has only the median. The second linked list has the three quartiles. And so on. In this ideal scenario there are about $\log n$ levels. This means that find is $O(\log n)$, as one essentially does binary search.

But what about insertion: one insertion would ruin the ideal properties described above. One could try to “rebalance” if the median drifted too far out. But a simpler approach is to only require the properties to hold approximately. And to use randomness.

C2.2 Implementation of Insertion

One uses randomness to determine the membership of the levels. Specifically:

\[
\textit{when inserting, toss a coin and add a node to the level above with probability 1/2 (and recurse)}
\]

So we only have expected $O(\log n)$ running times. (We will call it a Las Vegas algorithm.)

A natural implementation is to have an ordinary linked list at the bottom level. Each level above is a linked list with a down-pointer to the corresponding node in the lower level. To keep these levels all anchored, have a linked list of pointers to the start of each level, essentially acting as a collection of dummy nodes.
A simple if inelegant procedure for insert is to first see if the value is in the list (if one is prohibiting repeats). Then toss the coin(s) to determine how far up the levels it will go. And then start the insertion process in the topmost level.

Note that the randomness provides a good average run time, and a very low chance of bad behavior. And this is regardless of the actual data.

**Exercises**

1. Explain how one can add an int to each node that allows one to offer the rank operation. This method should return the value of specified rank in expected $O(\log n)$ time.
Chapter C3: Binary Search Trees

A *binary search tree* is used to store ordered data to allow efficient queries and updates.

### C3.1 Binary Search Trees

A *binary search tree* is a binary tree with values at the nodes such that:

*left descendants are smaller, right descendants are bigger.* (One can adapt this to allow repeated values.)

This assumes the data comes from a domain in which there is a *total order*: you can compare every pair of elements (and there is no inconsistency such as $a < b < c < a$). In general, we could have a large object at each node, but the object are sorted with respect to a *key*. Here is an example:

```
     53
   /   \
  31    57
 /     / \
12    34  56  69
|      |     |
5      68   80
```

### C3.2 Insertion and Deletion in BST

To find an element in a binary search tree, one compares it with the root. If it is larger, go right; if it is smaller, go left. And repeat. The following method returns `nullptr` if not found:

```c
find(key x) {
    Node *t=root;
    while( t!=nullptr && x!=t→key )
        t = ( x<t→key ? t→left : t→right );
    return t;
}
```

Insertion is a similar process to searching, except you need a bit of look ahead.

To remove a value from a binary search tree, one first finds the node that is to be removed. The algorithm for removing a node $x$ is divided into three cases:
• Node \( x \) is a leaf. Then just delete it.

• Node \( x \) has only one child. Then delete the node and do “adoption by grand-parent” (get old parent of \( x \) to point to old child of \( x \)).

• Node \( x \) has two children. Then find the node \( y \) with the next-lowest value: go left, and then go repeatedly right (why does this work?). This node \( y \) cannot have a right child. So swap the values of nodes \( x \) and \( y \), and then delete the node \( y \) using one of the two previous cases.

The following picture shows a binary search tree and what happens if 11, 17, or 10 (assuming replace with next-lowest) is removed.

All modification operations take time proportional to depth. In best case, the depth is \( O(\log n) \) (why?). But, the tree can become “lop-sided”—and so in worst case these operations are \( O(n) \).

C3.3 Rotations

In order to talk about “self-balancing” trees, we need the concept of rotations. A rotation can be thought of as taking a parent-child link and swapping the roles. Here is a picture of a rotation of \( B \) with \( C \):

C3.4 AVL Trees

There are several ideas to force a binary search tree to be balanced. One popular version is a red-black tree. But here we consider AVL trees. These are named after their inventors: Adelson-Velsky and Landis.
The idea of “self-balancing” is to add some property, contract, or invariant that is maintained. The property cannot be too close to balanced, since that would probably entail complete rebuilding of the tree too often. The idea is to come up with a condition for “nearly balanced” and such that if the tree becomes too far from balanced, simple operations make it very nearly balanced.

The invariant for AVL trees is:

\[
\text{for each node, the height of its left and right subtrees cannot vary by more than 1.}
\]

Each node keeps track of its height (that is, the height of the subtree rooted at it). This value can conceptually be updated after each insertion by going up the tree. (Only the ancestors of the new node can have changed.)

◊ **Insert in AVL**

We now consider how to perform the **insert**. As usual, there is a unique place in a binary search tree where a new node can be added while preserving the binary search tree property. So the new node is added there. If the result is a valid AVL tree, we are done. But what happens if it is not?

The tree was good just before the insertion. So the imbalance in the heights at any node can be at most 2. Consider the lowest node that is unbalanced. Let’s say it is \( g \). Assume that \( g \) has children \( c \) and \( k \), where node \( c \) has height 2 more than \( k \). Since the tree was good just before insertion, node \( c \) can have only one child that is too deep.

Now there are two possibilities. The first is that the deeper child of \( c \) is on the same side of \( c \) as \( c \) is of \( g \). Say the deeper child is \( b \), and \( b \) is the left child of \( c \) and \( c \) is left child of \( g \). Then rotation of the edge \( gc \) does the trick. The root of the subtree becomes \( c \), it becomes balanced, and its height is back to what it was before the latest insertion, so the whole tree is balanced.

Here is a picture of such a rotation. (You can think of node \( a \) as just having been inserted.)
The second possibility is that the deeper child of c is on the other side of c as c is of f. Say the deeper child is e, and e is the right child of c. Then a “rotation” of the whole gce does the trick: e becomes parent of both c and g. This can be programmed as two successive rotations—first ec then eg—and thus is often called a double rotation. The root of the subtree becomes e, it becomes balanced, and its height is back to what it was before the latest insertion, so the whole tree is balanced.

Here is a picture of such a rotation. (You can think of node d as just having been inserted.)

![Rotation Diagram]

♦ **Analysis**

The result is a running time of \( O(\log n) \). We claim! Okay: so one needs a theorem.

**Theorem.** Any AVL tree of height \( d \) has at least \( f_d \) nodes, where \( f_d \) is the \( d \)th Fibonacci number.

**Proof.** The proof is by induction. If we consider a semi-balanced tree of height \( d \), then one of its subtrees has height \( d - 1 \) and the other one has height \( d - 2 \) or \( d - 1 \). The first has at least \( f_{d-1} \) nodes. The second has at least \( f_{d-2} \) nodes. And so the whole tree has at least \( f_{d-1} + f_{d-2} = f_d \) nodes.

Since \( f_d \) is exponential in \( d \), this means that the height is \( O(\log n) \). One can also support the delete operation.

**Exercises**

1. Explain how one can add an int to each node that allows a binary search tree to offer the rank operation. This method should return the value of specified rank in time proportional to the depth of the tree
Chapter C4: Heaps and Binomial / Fibonacci Heaps

C4.1 Heaps

A heap is the standard implementation of a priority queue. A min-heap stores values in nodes of a tree such that

**heap-order** property: for each node, its value is smaller than or equal to its children’s

So the minimum is on top. (A max-heap can be defined similarly.)

The standard heap uses a binary tree. Specifically, it uses a complete binary tree, which we define here to be a binary tree where each level except the last is full, and in the last level nodes are added left to right. Here is an example:

```
    7
   / \
  24 19
 /   /
25 29 68 40
```

Now, any binary tree can be stored in an array using level numbering. The root is in cell 0, cells 1 and cells 2 are for its children, cells 3 through 6 are for its grandchildren, and so on. Note that this means a nice formula: if a node is in cell \( x \), then its children are in \( 2x+1 \) and \( 2x+2 \). Such storage can be (very) wasteful for a general binary tree; but the definition of complete binary tree means the standard heap is stored in consecutive cells of the array.

C4.2 Heap Operations

The idea for **insertion** is to: **Add as last leaf, then bubble up value until heap-order property re-established.**

```python
Insert(v)
add v as next leaf
while v < parent(v) {
    swapElements(v, parent(v))
    v = parent(v)
}
```
One uses a “hole” to reduce data movement. Here is an example of inserting the value 12:

![Diagram of a heap with a hole to insert value 12]

The idea for `extractMin` is to: *Replace with value from last leaf, delete last leaf, and bubble down value until heap-order property re-established.*

```
ExtractMin()
    temp = value of root
    swap root value with last leaf
    delete last leaf
    v = root
    while v > any child(v) {
        swapElements(v, smaller child(v))
        v = smaller child(v)
    }
    return temp
```

Here is an example of `ExtractMin`:

![Diagram of a heap after `extractMin`]

Variations of heaps include:

- *d*-heaps; each node has *d* children
- support of merge and other operations: leftist heaps, skew heaps, binomial heaps, Fibonacci heaps
C4.3 Binomial Heaps

We consider two interesting extensions of the heap idea: binomial heaps and Fibonacci heaps. The latter builds on the former.

Binomial heaps retain the heap-property: each parent is smaller than its children (we’re assuming min-heap). But they do away with the restriction to using a binary tree and also allow more than one root. Specifically, we will maintain:

\[ \text{a collection of heap-ordered trees.} \]

Again let us start by just supporting \text{insert} and \text{extractMin}. We get amortized constant time for \text{insert} and amortized O(log n) time for \text{extractMin} (cannot do better).

Allowing a collection of trees, means that \text{insertion} can be made trivial:

\begin{verbatim}
Insert
crea te a new heap
\end{verbatim}

Of course, this opting for lazy insertion adds work to \text{extractMin}. So, during that operation we will do extra computation to tidy up the heap.

In particular, the idea chosen by binomial heaps ensures that each execution of \text{extractMin} runs in time proportional to \( d + \log n \), where \( d \) is the number of \text{insert}'s since the previous \text{extractMin}. In essence, one can think of the lazy insertion operation putting aside a dollar for future computation.

Enough ideating. Here is the implementation. The tidy-up phase of the \text{extractMin} operation achieves the following situation: \text{there is at most one tree of each given root-degree.}

\begin{verbatim}
ExtractMin
calculate minimum root and store its value
add children of it as trees to the collection
while there are two trees of same root-degree do
    merge the two trees with larger-value root becoming child of smaller-value root
return stored value
\end{verbatim}

A simple way to implement the loop is to use an array with slots for a tree of each root-degree. By iterating through the roots, the loop takes time proportional to the
initial number of roots, which is at most $d + x$, where as above $d$ is the number of recent insert’s, and $x$ is the maximum root-degree at the beginning.

Now, all trees with root-degree zero are just isolated nodes. Also, all trees with root-degree $d$ are formed by combining two trees with root-degree $d - 1$. It follows (by induction) that every tree with root-degree $d$ has precisely $2^d$ nodes. In particular, the highest root-degree is at most $\log n$, and indeed all nodes have at most $\log n$ children.

So the only thing that remains is some implementation details. It is customary to make use of doubly-linked circular linked lists (DCL). A DCL is used to store the roots. Further, each node has a children counter and a pointer to a DCL of its children. The DCL data structure is used because it enables constant-time removal of a node, addition of a node, and merging of two DCLs.

C4.4 Decrease-Key and Fibonacci Heaps

◊ Delete and Decrease-Key

Now, any priority queue that supports decreaseKey can trivially be extended to handle delete as well: simply decrease the key to some value guaranteed to be the smallest, and then do extractMin. (This assumes the input to both operations is a pointer to the entry/node—if one needs to find the node as well, then the operation takes linear time anyway.)

So, it remains to support decreaseKey. In any heap-like structure, one implementation of this operation is: decrease the value, and then bubble the value up if smaller than parent, until it reaches the correct level. This can be achieved if one adds parent pointers to all the nodes. Since the trees all have depth at most $O(\log n)$, this means that we have decreaseKey in that time.

But the inventors of Fibonacci heaps provided a way to do decreaseKey in $O(1)$ amortized time, as we now show. We adapt the binomial heap.

◊ Potential Functions

A common way to keep track of the arithmetic for amortized analysis is to define a potential function. This allowance is adjusted by every operation, both by contributions to future computation, and then by actual computation that reduces it.

Here is it in action in Fibonacci heaps. To get decreaseKey to run in amortized constant time, the first idea is that decreaseKey is implemented by detaching that node from its parent and making it into a new tree for the collection.

This sounds like we are done. But this upsets the earlier analysis of extractMin, because now trees of the same root-degree can have varying sizes. The solution to this is to take action if the same node loses too many children...
In the final version, a node is marked if it loses a child. The potential of a Fibonacci heap is defined to be

\[ \Phi = t + 2m \]

where \( t \) is the number of trees/roots in the collection and \( m \) is the number of marked nodes. If a marked node loses a second child, then some rebalancing occurs.

The Fibonacci numbers now appear, as analysis of this adaptation shows that a node with root degree-\( d \) has size at least the \( d \)th Fibonacci number. This is approximately \( \Phi^d \), where \( \Phi = (1 + \sqrt{5})/2 \) is the golden ratio.

One still has to go back and check that the claimed amortized running times of all previous operations remain valid, et cetera

**Exercises**

1. Show that, in a binomial tree of height \( d \), the number of nodes at depth \( k \) is given by the binomial coefficient \( \binom{d}{k} \).

2. Suppose that 2018 entries are stored in a binomial heap. How many trees are there?
Chapter C5: Hash Tables

C5.1 Buckets and Hash Functions

The hash table is designed to do the unsorted dictionary ADT. It consists of:

1. an array of fixed size (normally prime) of buckets
2. a hash function that assigns an element to a particular bucket

There will be collisions: multiple elements in the same bucket. There are several choices for the hash function, and several choices for handling collisions.

Ideally, a hash function should appear “random”! A hash function has two steps:

1. convert the object to int.
2. convert the int to the required range by taking it mod the table-size

A natural method of obtaining a hash code for a string is to convert each char to an int (e.g. ASCII) and then combine these. While concatenation is possibly the most obvious, a simpler combination is to use the sum of the individual char’s integer values. But it is much better to use a function that causes strings differing in a single bit to have wildly different hash codes. For example, one might compute the sum

$$\sum a_i 37^i$$

where $a_i$ are the codes for the individual letters.

C5.2 Collision Resolution

◊ Chaining

The simplest method of dealing with collisions is to put all the items with the same hash-function value into a common bucket implemented as an unsorted linked list: this is called chaining. It can be shown that if one inserts $n$ elements at random into $n$ buckets, then the expected maximum occupancy of a bucket is $O(\log n)$.

The load factor of a table is the ratio of the number of elements to the table size. Chaining can handle load factor near 1.

Example. Suppose hashcode for a string is the string of 2-digit numbers giving letters (A=01, B=02 etc.) Hash table is size 7. Suppose we store:
An alternative to chaining is called *open addressing*. In this collision-resolution method: if the intended bucket \( h \) is occupied, then try another one nearby. And if that is occupied, try another one.

There are two simple strategies for searching for a nearby vacant bucket:

- **linear probing**: move down the array until find vacant bucket (and wrap around if needed): look at \( h, h+1, h+2, h+3, \ldots \)

- **quadratic probing**: move down the array in increasing increments: \( h, h+1, h+4, h+9, h+16, \ldots \) (again, wrap around if needed)

Linear probing causes *chunking* in the table, and open addressing likes load factor below 0.5.

The operations of search and delete become a bit more complex. For example, how do we determine if string is already in table? And deletion must be done by *lazy deletion*: when the entry in a bucket is deleted, the bucket must be marked as “previously used” rather than “empty”. Why?

### C5.3 Rehashing

If the table becomes too full, the obvious idea is to replace the array with one double the size. However, we cannot just copy the contents over, because the hash value is different. Rather, we have to go through the array and re-insert each entry.
One can show (amortized analysis again) that this does not significantly affect the average running time. The point is that one can spread the cost of the dynamic resizing. In particular, if the hash table was previously rehashed at size $d$ and we now rehash at size $2d$, then there have been $d$ insert’s since the previous rehash. Thus the $O(2d)$ cost of the rehash is constant per insert.

### C5.4 Other Applications of Hashing

Hash functions can be used for quick testing for duplicates: if the hash function is “random” then two different items will almost surely hash to a different value. This idea can be used to test in expected linear time whether a collection of integers from some bounded interval has duplicates.

### Exercises

1. [Add Me]
Chapter C6: Disjoint Set Data Structure

In implementing Kruskal’s algorithm we need to keep track of the components of our growing forest. This requires a data structure that represents an arbitrary collection of disjoint sets. It should support two operations:

- **find** tells one what set a value is in,
- **merge** combines two sets.

In the section on Kruskal’s algorithm, we had an array implementation that took time $O(1)$ for find but $O(n)$ for merge.

C6.1 Disjoint Forests

Here is another idea. Again store values in an array $A$. But this time, each set is stored as a rooted sub-tree according to the following scheme:

- If $A[i] = i$, then $i$ is the label of the set and the root of some sub-tree.
- If $A[i] \neq i$, then $A[i]$ is the parent of $i$ in some sub-tree.

For example, if the components are $\{1, 2, 3, 7\}$, $\{4, 6\}$, $\{5\}$, then the array might be

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

which represents the following sub-trees.

```
    7
   / \
 1   2
   |
    3
```

To determine the label of the set of a node, one follows the pointers up to the root. To combine two sets, one changes the one root to point to the root. Thus, merge runs in $O(1)$ time, if one is given the labels.

The find operation take time proportional to the depth of the sub-tree. And so we haven’t made any progress, yet. But there is a simple fix: While merging, make the root of the shorter one point to the root of the taller one. It can be shown that after a
series of $k$ merges the depth is at most $\log k$. (See exercise.) This means that the merge operation run in time $O(\log n)$. Note that one can keep track of the depth of a sub-tree. Applied to Kruskal, this gives an $O(a \log a)$ algorithm for finding a minimum spanning tree, since the initial sorting of the edges is now the most time-consuming part.

C6.2 Pointer Jumping

There is another simple improvement to reduce the order of the running times for the data structure. Namely, one uses the work done when following the pointers to the root. Specifically: when executing find, make all pointers traversed point to the root.

It can be show that the result provides an extremely slowly growing run-time for the data structure.

Exercises

1. Suppose in Kruskal’s algorithm we use the rooted-tree disjoint-set data structure for keeping track of components. If the nodes are A,B,C,D,E,F,G,H and the edges that Kruskal adds are in order AB, AC, DE, EF, AG, DH, GH, what does the final data structure look like:
   (a) without pointer jumping?
   (b) with pointer jumping?

2. In the rooted-tree disjoint-set data structure without pointer-jumping, show that a tree of depth $d$ has at least $2^d$ nodes.