# Lecture Notes for CSCI 311: Algorithms
# **Set 13-Data Structures**

Professor Talmage

March 6, 2024

---

## 1 Abstract Data Types

As we just saw with Huffman Codes, storing data in structured ways can make algorithms much more efficient. Of course, we need to understand how to do this, what algorithmic techniques allow us to keep our data storage efficient, and how we measure efficiency for a data structure, where we tend to repeat operations many times. To that end, we will step back for a moment from programming paradigms and talk about how we store and use data, and how we evaluate those aspects of data handling. We will briefly discuss some data types and structures you have seen before, then move to some more complex structures, as well as new analysis tools.

**Definition 1** (ADT)**.** An *Abstract Data Type (ADT)* is a set of operations, specifying input and output types, and a set of *legal* sequences of operation instances.

- An *instance* of an operation $OP$, written $OP(args, ret)$ is one invocation-return pair. For example. $Enqueue(5, -)$.
- The dash indicates that the operation does not return anything, and we use similar notation for those which take no arguments.
- We can classify different operations as *accessors*, *mutators*, or *mixed operations*, based on whether they return something about the current state, change the current state, or both.

**Definition 2** (Data Structure)**.** A *data structure* is a particular implementation of an ADT.

While it is not always made clear in common usage, this distinction is important. The ADT is the problem statement, defining correct behavior by specifying the interface or external behavior. A structure is a particular algorithm to satisfy that specification or solve that problem. We prove correctness of structures by proving their behavior is that the ADT specifies. Note that this is not the only way to specify data structures, but I personally favor it as it makes correctness proofs relatively straightforward–since everything is a sequence of operation instances, we can inductively prove that the sequence of return values the algorithm generates is correct.

**Example:**

**Definition 3** (Queue)**.** A *queue* over values $V$ is a data types with two operations:

1. *Enqueue* accepts one value $x \in V$ as argument, returns nothing: $Enqueue(x, -)$
2. *Dequeue* accepts no argument, returns one value $x \in V$: $Dequeue(-, x)$

A sequence $\rho$ of *Enqueue* and *Dequeue* instances is legal iff every Dequeue instance returns the argument of the first previous *Enqueue* instance in $\rho$ whose argument has not already been returned by a *Dequeue* instance. If no such *Enqueue* instance exists, *Dequeue* returns the special value $\perp$, indicating empty queue.

In this style of definition, state is implicit, defined by the sequence of past operations. This is sufficiently descriptive, as determinism implies that following the same steps will yield the same state, and is convenient for proving an algorithm correct. However, it is not necessarily easy to compare two different sequences to determine whether they end in equivalent states. For contrast, we could also define a queue by state:

**Definition 4** (Queue, alt.)**.** A *queue* is a data type which stores an ordered list of elements passed as arguments to *Enqueue*. Each *Enqueue* adds elements to the same end, known as the tail. Each *Dequeue* instance removes and returns one element in the queue in the order they were added by returning the element at the head, the end opposite the tail.

The danger here, is that this definition is dangerously close to specifying an implementation, not just an interface. Any state-based definition will need to describe internal state, and the point of an ADT is to allow different implementations of the same behavior.

**Aside:** You may wonder what happened to the *Peek* operation. A fundamental Queue type does not have a *Peek* operation. We can *augment* a queue with a peek by adding it to the operation set and defining which sequences with *Peek* instances are legal (they return the oldest un-*Dequeue*d element in the queue). This is a different ADT, and the difference is sometimes important. For example, in distributed systems, adding a *Peek* infinitely increases the power of the type to solve other computation problems.

**Exercise:** Draw the states of a queue at each step of the following sequence of operation instances:

$$Deq(-,\perp)\cdot enq(17)\cdot Enq(10)\cdot Enq(18)\cdot Deq()\cdot Enq(4)\cdot Deq()\cdot Deq()\cdot ()$$

Use the notation $h[x,y,\ldots,z]t$, where the oldest element is at the end labeled $h$ for “head” and the newest element is at the end labeled $t$ for “tail”.

**Exercise:** Looking at the sequence-based specification for a Queue, what would change to specify a stack, instead?

**Exercise:** Give a sequence-based specification for priority queue.

**Definition 5** (Priority Queue)**.** A *priority queue* over value set $V$ provides operations

1. *Insert*: Accepts one value, no return: $Insert(x,-)$
2. *Delete*: no input, returns one value: $Delete(-,x)$

A sequence of *Insert* and *Delete* operation instances is legal iff each *Dequeue* returns the argument of a previous *Insert* which has not already been returned by a *Delete*, with highest (or lowest) priority, as specified by some priority function $p(x) : V \to \mathbb{R}$.

## 2 Data Structures

We will now set aside data types and talk about data structures. That is, we will consider how to implement various types efficiently, not worry so much about what those types are doing. To specify a data structure, we have to decide how we arrange the data in memory and how we represent the relationships among the data. Practically, we provide code for each operation in the ADT, as well as potentially any helper functions we need internal to our implementation. To argue the correctness of a data structure, we must prove that the code makes each function return the value specified by the ADT. For efficiency, we talk about the runtime of each operation separately.

**Exercise:** What is your favorite queue implementation? Verbally describe how you implement *Enqueue* and *Dequeue*, how you know they are correct, and their efficiencies.

## 2.1 Trees

While there are interesting things to explore related to other data types and structures, we particularly want to focus on trees. As you likely know, trees are a recursive, hierarchical structure which appear everywhere in computer science (we just used them to represent binary encodings!). While you have probably used trees before, we will consider some more complex variants and the benefits they can have.

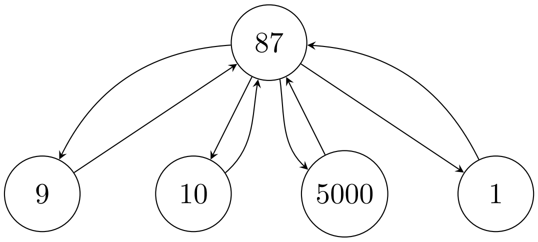
**Definition 6** (Rooted Tree)**.** A *rooted tree* is a collection of *nodes*. Each node has the following attributes:

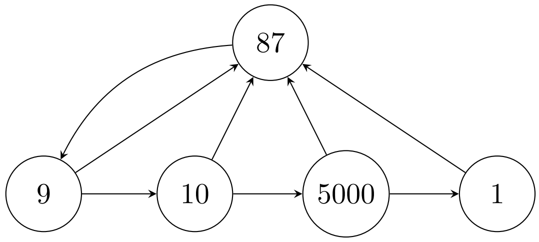
- *parent*: another node
- *children*: A list of other nodes
- *value*: data contained in this node

Properties:

- Trees are connected and acyclic.
- If the state of the tree does not change between calls, $x.parent = y$ iff $x \in y.children$.
- Exactly one node has $x.parent = x$. Call this node the root. (Alternately, can have $root.parent = \perp$.)
- Nodes with $children = []$ are called *leaves*

**Aside:** This is a parent-child representation of a tree. This is perhaps the most common and intuitive representation, capturing the recursive structure, as each node is the root of its own subtree. Another commonly-useful implementation is "Left-Child, Right-Sibling". Each node tracks only its parent, left child, and right sibling. To get all the children of a given node, go to its left child, then follow right-sibling pointers until one is null. The advantage is that each node is constant size, containing references to exactly three other nodes, no matter how many children it has.





Remember, this is a structure definition, not an ADT specification. We are defining implementation details, not the functions and behavior a user will see. Trees are useful for implementing a wide variety of ADTs, but we are going to focus on how to build different types of trees, more than the ADTs they satisfy.

## 2.2 Binary Search Trees

**Definition 7** (Binary Tree)**.** A *binary tree* is a rooted tree where every node has at most two children.

- This constraint eliminates the problem of varying node size.
- We refer to *left* and *right* children of a node.

**Definition 8** (BST)**.** A *binary search tree* is a binary tree that satisfies the binary search tree property:

- For every node $y$ in the left subtree of node $x$ and every node $z$ in the right subtree of $x$, $y.value \leq x.value \leq z.value$.

**Exercise:** Place the following values in a binary tree s.t. it is a BST:

$$\{18, 7, 4, 5, 13, 20, 6, 1, 2, 14\}$$

Now draw another, different BST containing the same values.

**Operations:** BSTs typically provide at least these operations. Depending on the ADT you wish to implement, you may add others or change whether some of these are private functions.

Mutators:

- `BST-Insert`
- `BST-Delete` (mixed operation)

Accessors:

- `BST-Search`
- `BST-Min`
- `BST-Max`
- `BST-Successor`(*node*)

**Implementations:**

**Exercise:** Give verbal descriptions of `BST-Insert` and `BST-Search` implementations. What would change if your tree was in left-child, right-sibling representation?

**Exercise:** What concern makes `BST-Delete` trickier to implement?

**Definition 9** (Successor)**.** In a BST $T$, the *successor* of a node $x$ is the node in $T$ with the smallest value larger than $x$.

To find a node's successor, we must consider three cases:

1. If $x.rightChild$ exists, then $successor(x) =$ `BST-Min`$(x.rightChild)$
2. Else, if $x = x.parent.leftChild$, then $successor(x) = x.parent$.
3. Else, follow parent links from $x$ until an ancestor is a left child and return that node's parent. If none such exists, $x$ is the largest node in the tree and has no successor.

To delete a node, we need to maintain the search tree structure. First, we cannot leave the tree disconnected, so if we delete an internal node, we need to replace it with another node in the tree. Second, we must maintain the BST Property, so the replaced node must have the same relation between its value and other nodes. We do this by replacing the deleted node $x$ with its successor, since the successor is larger than anything smaller than $x$ and smaller than anything else in the tree larger than $x$. We actually are only concerned about the first of the cases for finding the successor in deleting, since if $x$ has zero or one children, we can patch the tree more simply.

`BST-Delete`($x$):

1. If $x$ is a leaf, delete it directly and return.
2. If $x$ has one child, move that child to $x$'s place by updating the child's parent pointer and $x$'s parent's child pointer.
3. If $x$ has two children, replace $x$ with $successor(x)$ by recursively deleting $successor(x)$ and replacing $x$'s value without deleting its node.

Note that since $x$ has two children when we want its successor, we know that the successor is the smallest value in $x$'s right subtree. Further, $successor(x)$ will not have a left child, so deleting $successor(x)$ will not lead to a search for another replacement. This means that `BST-Delete` completes in a single pass down the tree.

**Exercise:** How do we know that $successor(x)$ has no left child?