

# Thoughts on Project 6

## Queues

### Introduction to Trees

---

CS 311 Data Structures and Algorithms  
Lecture Slides  
Friday, October 30, 2020

Glenn G. Chappell  
Department of Computer Science  
University of Alaska Fairbanks  
[ggchappell@alaska.edu](mailto:ggchappell@alaska.edu)  
© 2005–2020 Glenn G. Chappell  
Some material contributed by Chris Hartman

---

# Thoughts on Project 6

## Thoughts on Project 6

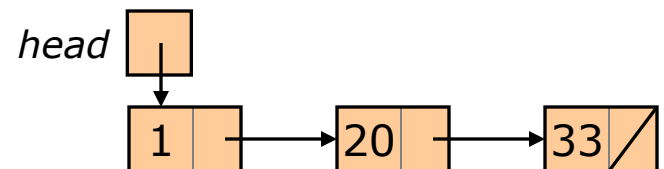
### Overview

Project 6 has two exercises, both based on our Linked List node that contains a smart pointer.

- In Exercise A, you will write a function template that reverses a Linked List in linear time, in-place, using *no* value-type operations.
- In Exercise B, you will write a class template that holds an **associative dataset** in a Linked List.

The node definition is in the revised Linked List header file written in class: `llnode2.h`. Your code should include this header.

In the following slides, I draw Linked Lists as usual. However, the arrows represent `std::unique_ptr`, not plain pointers.

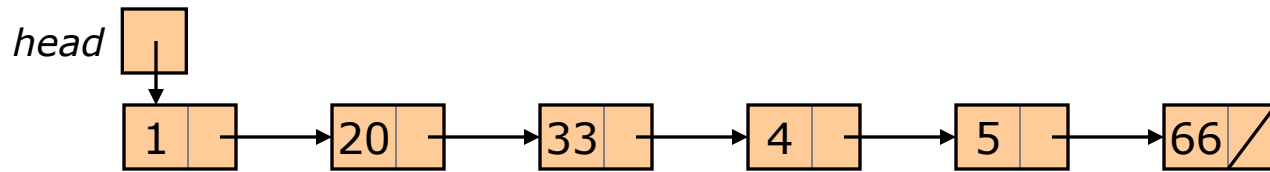


And the slash represents an **empty** `unique_ptr`, which plays the same role as a null pointer.

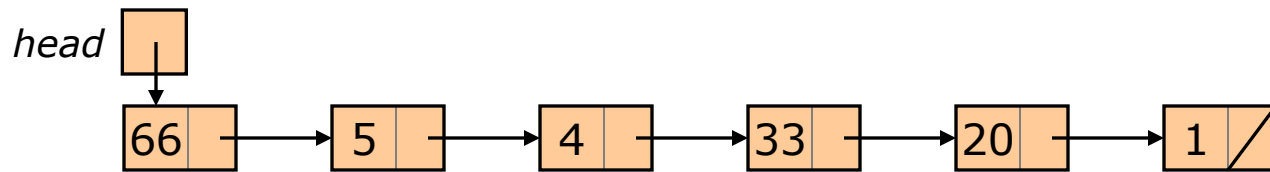
## Thoughts on Project 6

### Reversing a Linked List [1/4]

To **reverse** a Linked List means to start with a list like this:



... and end with this:



We can reverse a Linked List using code that:

- Runs in linear time.
- Is in-place—so no buffers, no copies of the list, and no recursion.
- Uses *only* pointer manipulation—no value-type operations, and no new nodes created.

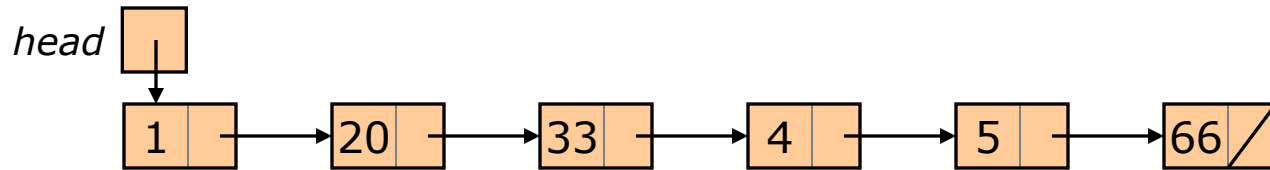
We look at how to write such code.

## Thoughts on Project 6

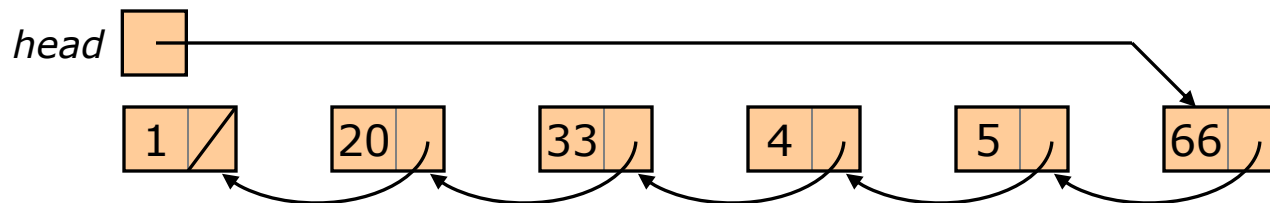
### Reversing a Linked List [2/4]

Pictures can be deceiving. Below we show the reverse operation again, but we leave each node in the same place in the picture.

Start with a Linked List like this:



... and end with this:

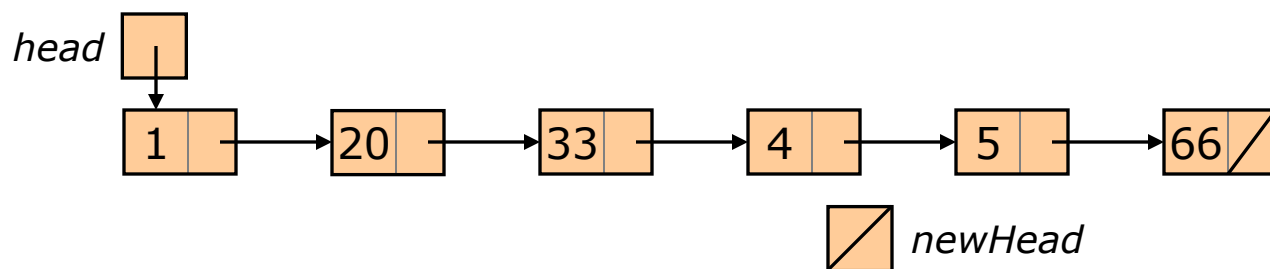


Observe that the only things that change are pointers.

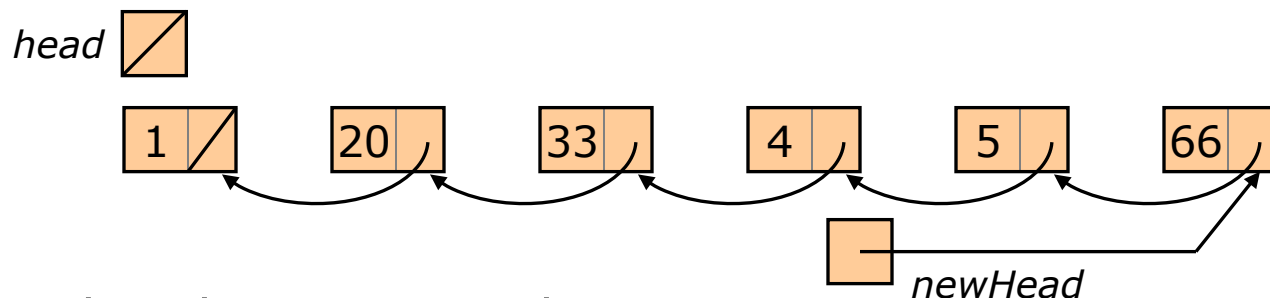
## Thoughts on Project 6

### Reversing a Linked List [3/4]

The code must make a new list from existing nodes. *newHead* will point to this new list, which holds nodes that have been reversed. *head* points to the nodes that are not reversed yet. Here is the situation at the start:



And here is the situation after we have reversed everything:

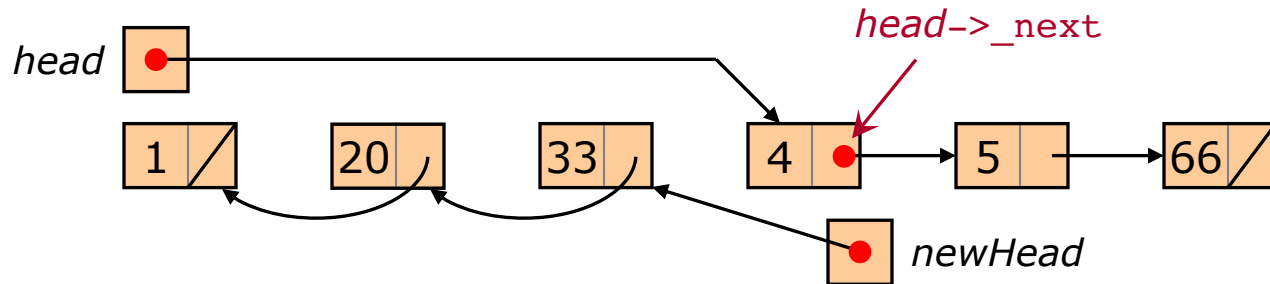


To finish, set *head* = *newHead*. ← Any observations? (Remember: *unique\_ptr*.)

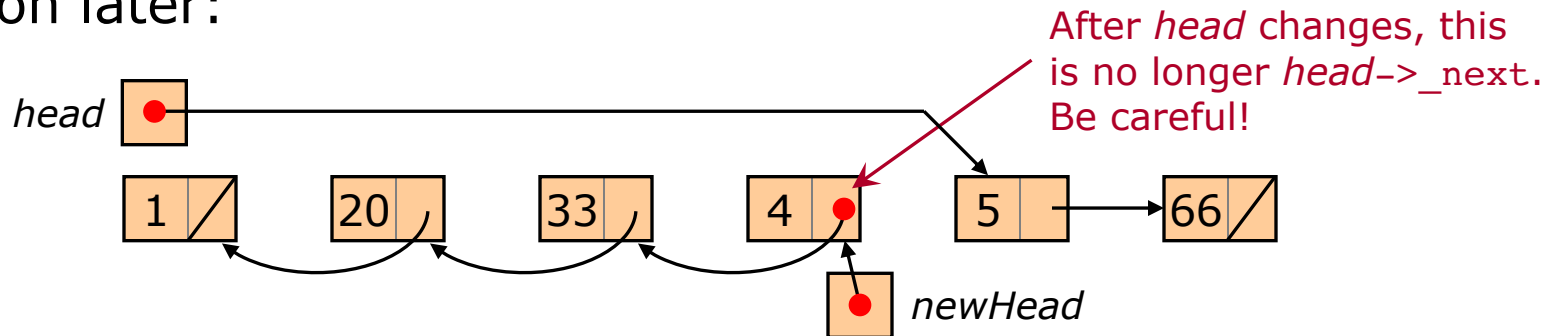
## Thoughts on Project 6

### Reversing a Linked List [4/4]

Between start & finish, have a loop. What should one iteration do?  
Here is an intermediate state:



One iteration later:



Q. What does one iteration do?

A. Rotate 3 pointers: *newHead*, *head*, *head->\_next*.

Loop until *head* is empty: `while (head) { ... }`

Above, these three are marked with red dots.

## Thoughts on Project 6

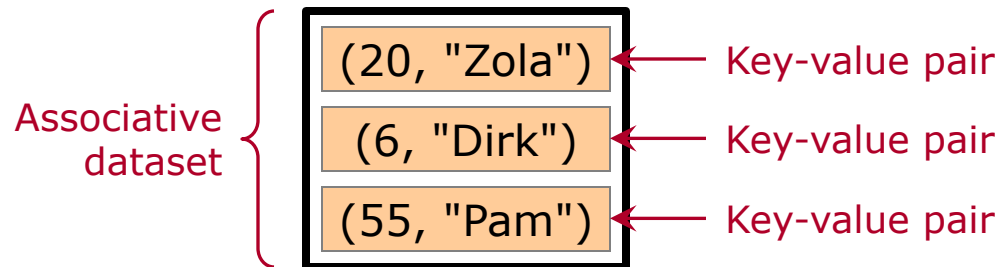
### Associative Dataset [1/4]

An **associative dataset** is a collection of items, each of which has a **key**, by which it can be looked up.

Often there is a **value** associated with each key.

| Key | Value |
|-----|-------|
| 20  | Zola  |
| 6   | Dirk  |
| 55  | Pam   |

An associative dataset can be viewed as a collection of **key-value pairs**. Typically (but not always) it is stored that way.



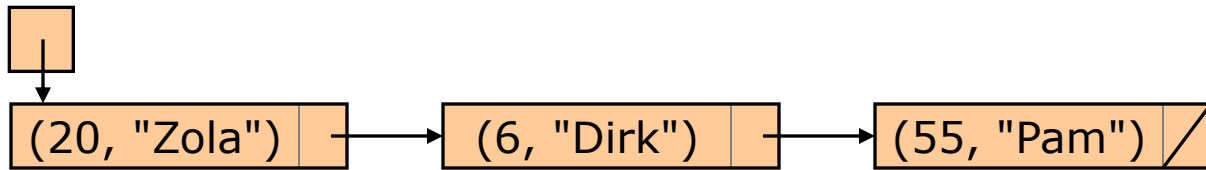


## Thoughts on Project 6

### Associative Dataset [2/4]

---

You will write a class that holds an associative dataset. It will have a single data member: a `unique_ptr` to a Linked List node. So the dataset will be stored in a Linked List. Each node holds one key-value pair.



The key type and value type will be specified by the client code.

Duplicate keys are not allowed. More precisely, the keys in two different nodes will never compare equal (`==`).

Values in the dataset are modifiable. Keys should not be—but a key may be replaced by removing it and inserting a new key.

## Thoughts on Project 6

### Associative Dataset [3/4]

---

Some of the operations available on a dataset:

- **find.** Given a key. Returns a pointer to the associated value, or `nullptr` if the key is not in the dataset.
- **insert.** Given key & value. Inserts the pair into the dataset. If an existing pair has an equal key, then the new pair replaces it.
- **erase.** Given a key. Removes item with this key, if any.
- **traverse.** Given a function (object). Calls it on each key-value pair.
- **size.** Returns the number of items.
- **empty.** Returns `true` if there are no items.

Your implementation should be reasonably efficient, within the limitations of the project requirements.


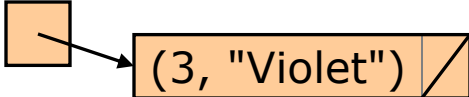
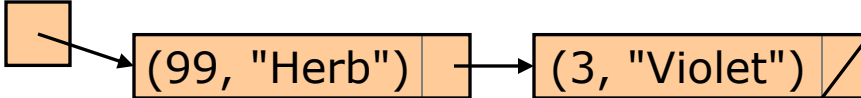
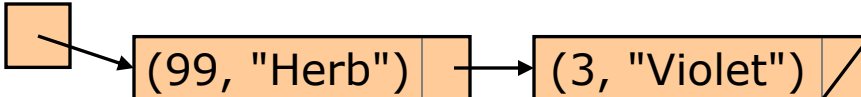
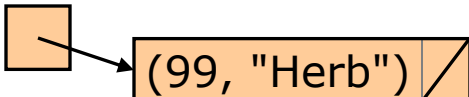
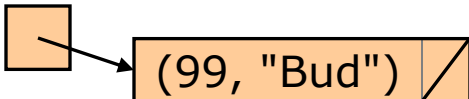
However, these requirements make truly high efficiency impossible. A Linked List of key-value pairs is a poor way to store an associative dataset.

Later in the semester, we will look at better implementations.

# Thoughts on Project 6

## Associative Dataset [4/4]

### Example

- Start: empty dataset. 
- insert(3, "Violet") 
- insert(99, "Herb") 
- erase(5) 
- erase(3) 
- insert(99, "Bud") 

---

# Review

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.

# Unit Overview

## Data Handling & Sequences

---

### Major Topics

- ✓ ■ Data abstraction
  - ✓ ■ Introduction to Sequences
  - ✓ ■ Interface for a smart array
  - ✓ ■ Basic array implementation
  - ✓ ■ Exception safety
  - ✓ ■ Allocation & efficiency
  - ✓ ■ Generic containers
  - ✓ ■ Node-based structures
  - ✓ ■ More on Linked Lists
  - ✓ ■ Sequences in the C++ STL
  - ✓ ■ Stacks
  - Queues
- 
- Smart Arrays
- Linked Lists

# Review Stacks

**Stack:** another container ADT.  
Restricted version of Sequence:  
**Last-In-First-Out (LIFO).**

Three primary operations:

- **getTop**
- **push**
- **pop**

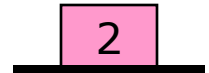
A Stack can be implemented simply  
as a wrapper around some  
existing Sequence type.

The STL has `std::stack`—a  
**container adapter**. You can  
choose the container. Default:  
`std::deque`.

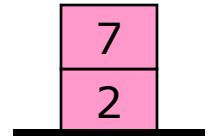
1. Start:  
empty Stack.



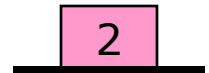
2. Push 2.



3. Push 7.



4. Pop.



5. Pop.  
Stack is empty again.



6. Push 5.



---

# Queues



# Queues

## What a Queue Is — Idea [1/2]

---

Our fourth ADT is **Queue** (say “Q”). This is yet another container ADT; that is, it holds a number of values, all the same type.

A Queue is ...

- ... very similar to a Stack in *definition*,
- ... rather different from a Stack in *implementation*, and
- ... very different from a Stack in *application*.

# Queues

## What a Queue Is — Idea [2/2]

A *Queue* allows **First-In-First-Out (FIFO)** access to data.

- What we do with a Queue:
  - **Enqueue** (say “N Q”): add a new value at the *back*.
  - **Dequeue** (say “D Q”): remove the value at the *front*.
- The first item added is the first removed.
  - Think of people standing in line. (This is also a good way to remember which end is the *front* and which is the *back*.)


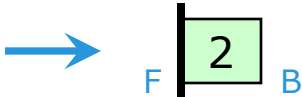
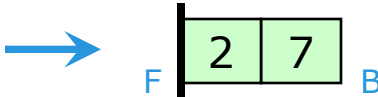

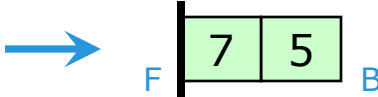
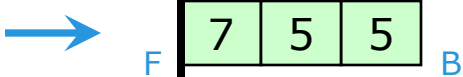



Some people use “FIFO” as another name for a Queue.

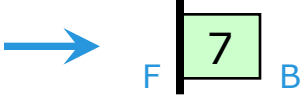
A Queue is another restricted version of a Sequence.

- We can only insert at one end and remove at the other.
- We (typically) cannot iterate through the contents.

# Queues

## What a Queue Is — Illustration

1. Start: an empty Queue. 
2. Enqueue 2. 
3. Enqueue 7. 
4. Dequeue. 
5. Enqueue 5. 
6. Enqueue 5. 
7. Dequeue. 
8. Dequeue. 
9. Dequeue. 

Queue is empty again.
10. Enqueue 7. 

*Compare this with Stack!*

# Queues

## What a Queue Is — Waiting

---

Conceptually, a Queue carries out the idea of **waiting in line**.

- Items that need to be processed are enqueued.
- When we are able to process an item, we dequeue it and process it.
- As long as the processor keeps going, no item waits forever. They are all processed eventually.

In practice, nearly every use of a Queue has this idea behind it.

# Queues

## What a Queue Is — ADT

---

### ADT **Queue**

- **Data**
  - A finite sequence of data items, all the same type. One end is the **front**; the other end is the **back**.
- **Operations**
  - **getFront**. Look at front item.
  - **enqueue**. Add a new item at the back.
  - **dequeue**. Remove front item.
  - To avoid errors we need information about the number of items:
    - **isEmpty**. Return true if Queue is empty.
    - **size**.
  - And the usual:
    - **create**.
    - **destroy**.
    - **copy**.

Three primary operations  
(retrieve, insert, delete)



# Queues

## Implementation — Sequence Wrapper

---

Like a Stack, a Queue can be a wrapper around a Sequence.

- We need fast insertion at one end and fast removal at the other.
  - NOT a resizable array.
  - A Doubly Linked List works.
  - A Singly Linked List works, *if* it has the right interface. We need to maintain an iterator to the last item. We can then insert at the end and remove at the beginning. Since we never do remove-at-end, we can always update the iterator when it changes.
  - Something like `std::deque` works.
- As with Stacks, the Queue operations are typically implemented in the underlying Sequence container. We only need to write a few one-line wrapper functions.

Implementing a Queue can be trickier than implementing a Stack.

- Making a Stack out of a resizable array is easy. But a Queue implemented analogously is inefficient, as noted above.

Next, a different way to implement a Queue.

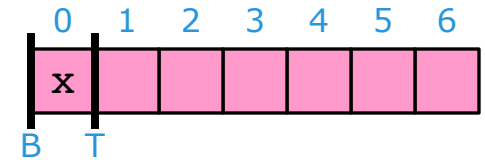
# Queues

## Implementation — Possibility: Array + Markers

Another idea: store the data in an array with extra space. Markers indicate the beginning and end of the actual data.

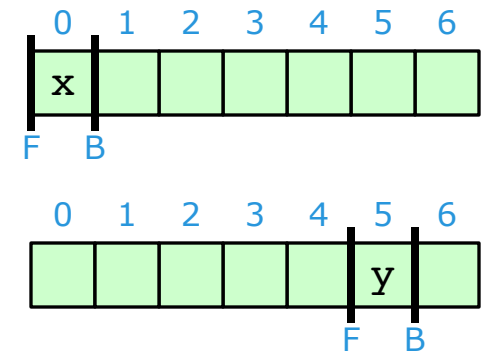
Consider a Stack based on this idea.

- Begin with 1 item ('x') stored at index 0.
- `push('y')` 5 times, then `pop()` 5 times.
- Result: Exactly what we began with.



Consider a Queue based on this idea.

- Begin with 1 item ('x') stored at index 0.
- `enqueue('y')` 5 times, then `dequeue()` 5 times.
- Result: 1 item ('y') stored at index 5.
- If the array is as shown, two more enqueue operations make the data "crawl" off the end.



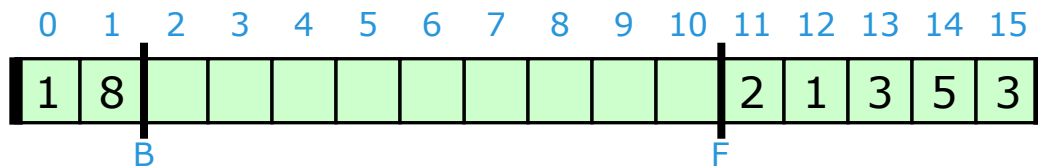
This is another reason Queues can be trickier to implement than Stacks. How can we deal with this problem?

# Queues

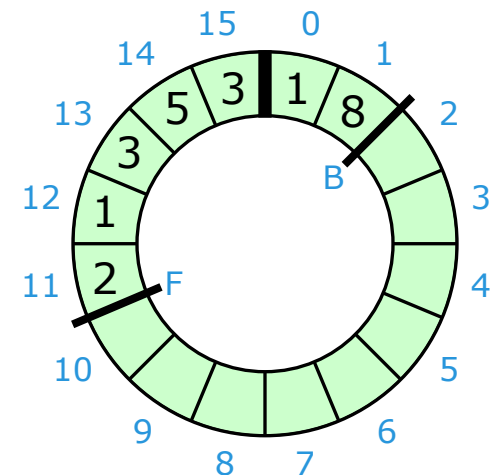
## Implementation — Circular Buffer [1/2]

When we store a Queue in an array with markers, we can deal with the problem of “crawling data” using a **circular buffer**.

- A circular buffer is just an ordinary Sequence. However, we think of the ends as being joined.
- Markers indicate the front and back of the Queue.
- If possible, when the Queue expands or contracts, we avoid resizing the underlying Sequence, and merely move the markers.



Physical Structure



Logical Structure



# Queues

## Implementation — Circular Buffer [2/2]

What is the order of each of the following operations for a Queue implemented using a circular buffer stored in a resizable array?

- **getFront**
  - Constant time.
- **dequeue**
  - Constant time.
- **enqueue**
  - Linear time (reallocation may be required).
  - Constant time if no reallocation is required.
  - With a good reallocation scheme: amortized constant time.
- **isEmpty**
  - Constant time.
- **copy**
  - Linear time.

As (nearly) always.



# Queues

## In the C++ STL — Introduction

---

The STL has a Queue: `std::queue (<queue>)`.

`queue` is another *container adapter*: wrapper around a container.

Again, you get to pick what that container is.

`std::queue<T, container<T>>`

- `T` is the value type.
- `container<T>` can be any standard-conforming container with member functions `front`, `push_back`, `pop_front`, `empty`, and `size`, along with comparison operators.
- In particular, `container` can be `deque` or `list`.
- But *not* `vector` or `basic_string`, which have no `pop_front`.

`container` defaults to `std::deque`.

`std::queue<T> // = std::queue<T, std::deque<T>>`

# Queues

## In the C++ STL — Operations

The `std::queue` interface for the various ADT operations:

| ADT Operation | Implementation                     |
|---------------|------------------------------------|
| enqueue       | Member function <code>push</code>  |
| dequeue       | Member function <code>pop</code>   |
| getFront      | Member function <code>front</code> |
| isEmpty       | Member function <code>empty</code> |
| size          | Member function <code>size</code>  |
| create        | Default constructor                |
| destroy       | Destructor                         |
| copy          | Copy/move operations               |

← This one is different from `std::stack`, which has `top`.

`std::queue` also has:

- Member function `swap`.
- The various comparison operators (`==`, `<`, etc.).

# Queues

## Applications

---

Queues mediate many kinds of *communication*, particularly those involving *requests* for some kind of service.

### Usage

- Each new message or request is enqueued.
- To receive a message or process a request, dequeue it.

### Examples

- Print jobs are sent to a networked printer from multiple computers. Jobs are stored in a *print Queue*. When a computer has something to be printed, it enqueues the job. When the printer is available, it dequeues a job and prints it.
- In an application with a **Graphical User Interface (GUI)**, user input is generally received in the form of *events*. An event might be a mouse button-down or button-up, a keypress, part of a window being uncovered and needing redrawing, a timer expiring, or receiving a file over a network. Events will be stored in a Queue.

## Unit Overview

### The Basics of Trees

---

This ends our coverage of Sequences & restricted versions thereof.

Next we begin a short unit covering a very different basis for data structures: *trees*.

#### Major Topics

- Introduction to Trees
- Binary Trees
- Binary Search Trees

This unit covers only basic concepts. Some of the fancier kinds of trees will be covered later.

After this unit, we look at ADTs Table & Priority Queue.  
Implementations of these will mostly involves trees.

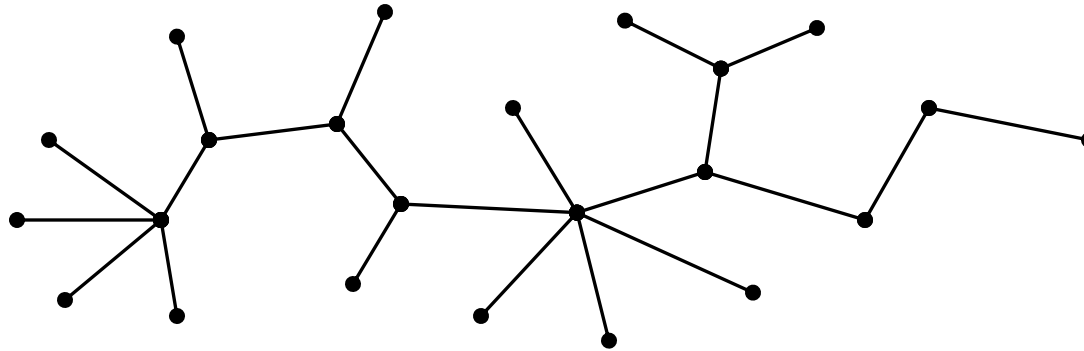
---

# Introduction to Trees

# Introduction to Trees

## What a Tree Is

A **tree** is a structure along the lines of the following.



Each dot is a **vertex** (Latin plural "**vertices**") or a **node**.

- I generally use *vertex* for the element of the tree as an abstract entity, and *node* for its representation in a computer program.

Each line is an **edge**. An edge joins two vertices.

A structure consisting of a collection of vertices, along with edges, each of which joins two vertices, is called a **graph**. (This is *not* the same meaning of “graph” that you used in algebra class.)

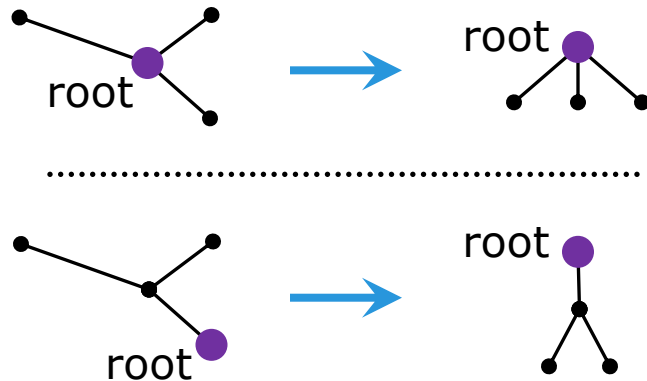
A **tree** is a **connected** (all one piece) graph containing no **cycles**.

# Introduction to Trees

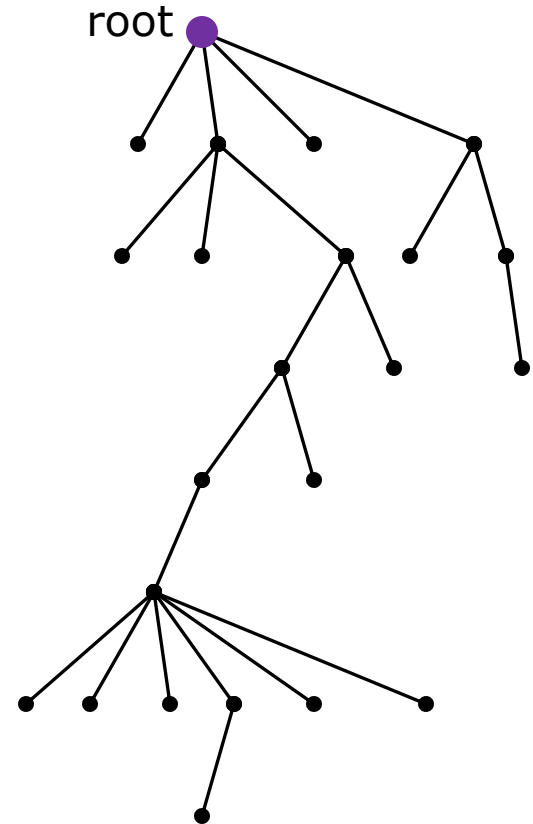
## What a Rooted Tree Is

A **rooted tree** is a tree with one vertex designated as the **root**.

When we *draw* a rooted tree, we will place the root at the top. Each non-root vertex hangs from some other vertex.



We will use **tree** to mean **rooted tree**.  
(We will have occasion to talk about non-rooted trees very near the end of the semester, but not before then.)

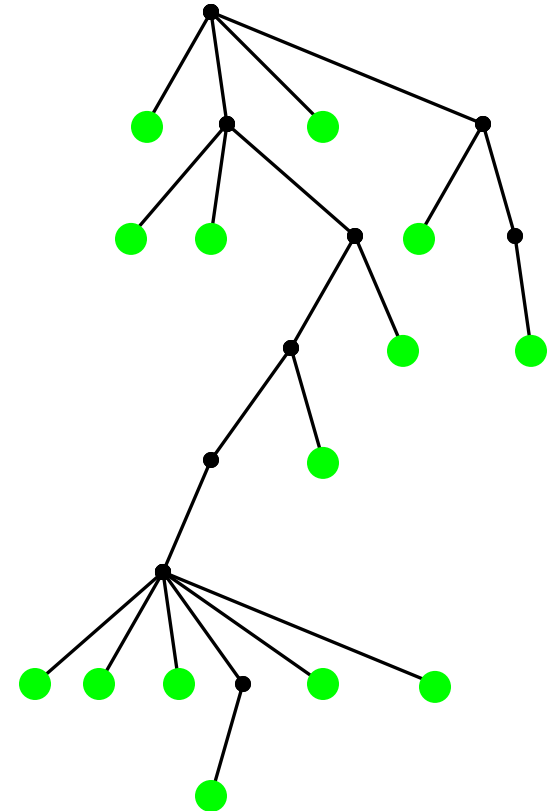




## Terminology for Rooted Trees [1/5]

Some of the terminology for rooted trees comes from plants\*.

- *Root* is an example.
- Another: a vertex with nothing hanging off of it is called a **leaf**.
- Think: what is true if a tree has just one vertex?



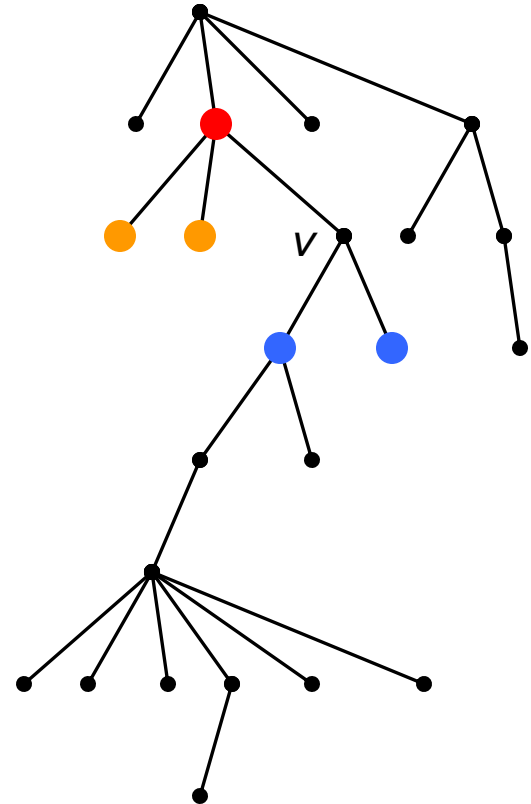
\*Upside-down plants, apparently.

# Introduction to Trees

## Terminology for Rooted Trees [2/5]

Other terminology comes from family trees.

- To illustrate this, we label a vertex  $v$  in the tree at right.
- The vertex that  $v$  hangs from is the **parent** of  $v$ .
  - Every vertex except the root has exactly one parent.
- The vertices that hang from  $v$  are the **children** of  $v$ .
  - A vertex with no children is a leaf.
- The other children of  $v$ 's parent are the **siblings** of  $v$ .

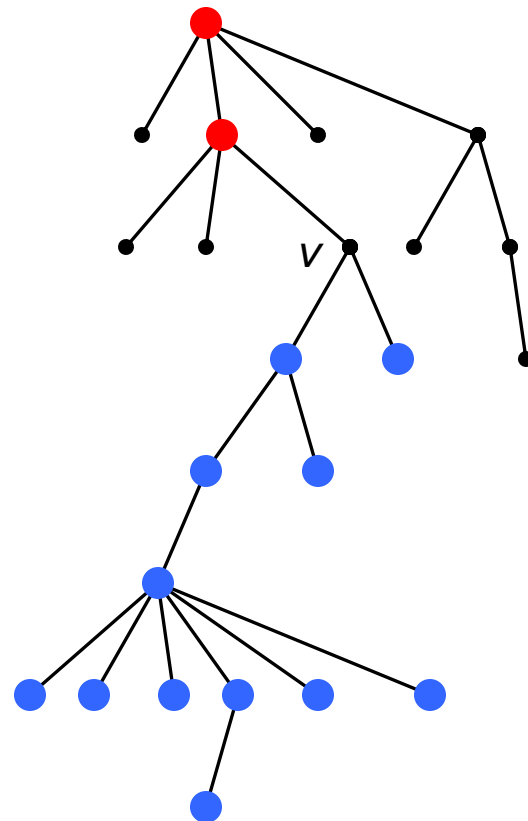


# Introduction to Trees

## Terminology for Rooted Trees [3/5]

The parent of  $v$ , and its parent, and its parent, etc., are the **ancestors** of  $v$ .

The children of  $v$ , and their children, and their children, etc., are the **descendants** of  $v$ .



# Introduction to Trees

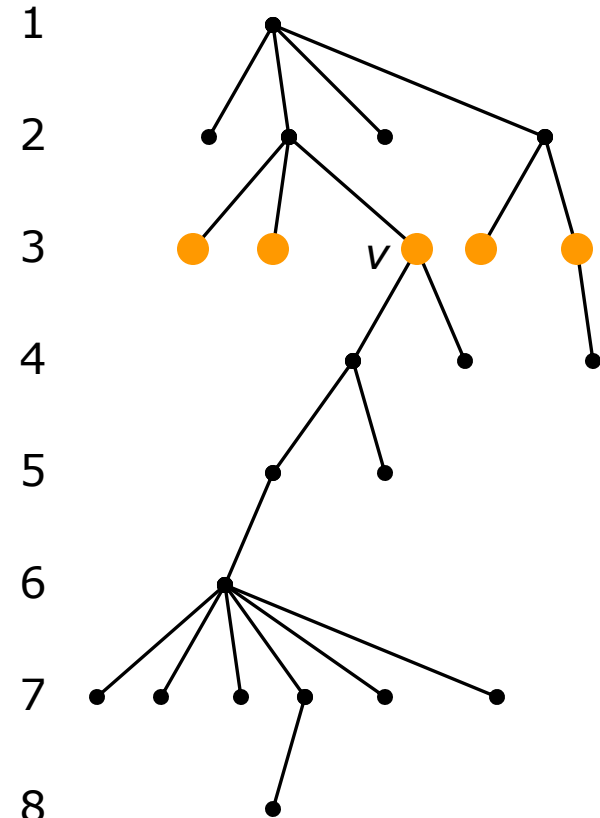
## Terminology for Rooted Trees [4/5]

The vertices of a rooted tree come in **levels**.

- The root is at level 1.
- Each other vertex has a level 1 greater than its parent.
- In the tree at right, level 3, which includes  $v$ , is drawn in orange.
- We *often* draw vertices in the same level as a horizontal row.

The **height** of a tree is the number of levels it has.

- The tree shown has height 8.
- Note. *Height* is sometimes defined as 1 less than this—the number of gaps between levels. We will not use this definition.



# Introduction to Trees

## Terminology for Rooted Trees [5/5]

A **subtree** consists of a vertex and all its descendants.

- Given a node  $n$ , the **subtree rooted at  $n$**  consists of  $n$  and all its descendants.
- Given a node  $n$ , a **subtree of  $n$**  is a subtree rooted at some child of  $n$ .
- Shown in green is a subtree of  $v$ . It is the subtree rooted at  $v$ 's child  $w$ .

