Queues
Trees

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, April 8, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

Unit Overview
Handling Data & Sequences

Major Topics

- ✓ Data abstraction
- ✓ Introduction to Sequences
- ✓ Smart arrays
  - ✓ Array interface
  - ✓ Basic array implementation
  - ✓ Exception safety
  - ✓ Allocation & efficiency
  - ✓ Generic containers
- ✓ Linked Lists
  - ✓ Node-based structures
  - ✓ More on Linked Lists
- ✓ Sequences in the C++ STL
- ✓ Stacks
- Queues

Review: Stacks in the STL (1/2)

The STL has a Stack: **`std::stack`**, in **`<stack>`**.

- STL documentation does not call **`std::stack`** a "container", but rather a "container adapter".
- This is because **`std::stack`** is explicitly a **wrapper** around some other container.

You get to pick what that container is.

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

- "**`T`**" is the value type.
- "*container*" can be **`std::vector`**, **`std::deque`**, or **`std::list`**.
- "*container*`<T>`" can be **any** standard-conforming container with member functions **`back`**, **`push_back`**, **`pop_back`**, **`empty`**, **`size`**, along with comparison operators (==, <, etc.).

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

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

Review: Stacks in the STL (2/2)

`std::stack` implements the various ADT operations as follows.

ADT Operation	**What to Call**
Push	Member function **`push`**
Pop	Member function **`pop`**
GetTop	Member function **`top`**
IsEmpty	Member function **`empty`**
Create	Default constructor
Destroy	Destructor
Copy	Copy constructor, copy assignment

`std::stack` also has member function size, which returns the size of the Stack, and the various comparison operators (==, <, etc.).

Review: Queues
What a Queue Is — Idea [1/2]

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

- Say “Q”.
- A Queue is …
  - … very similar to a Stack in **definition**,
  - … somewhat different from a Stack in **implementation**, and
  - … very different from a Stack in **application**.

Review: Queues
What a Queue Is — Idea [2/2]

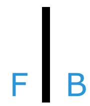
A *Queue* is a First-In-First-Out (FIFO) structure.

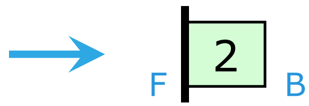
- What we do with a Queue:
  - **Enqueue**: add a new value at the *back*.
    - Say “N Q”.
  - **Dequeue**: Remove a value at the *front*.
    - Say “D Q”.
- The first item added is the first removed.
  - Think of people standing in line. (This is also a good way to remember which end is “front” and which is “back”.)
- Some people use other words for “enqueue” & “dequeue”.
  - “push” and “pop”, for example.

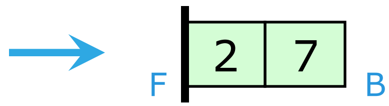
Thus, a Queue is another restricted version of a Sequence.

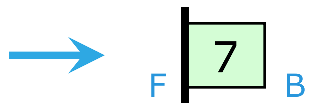
- We can only insert at one end and remove at the other.
- We (usually) cannot iterate through the contents.

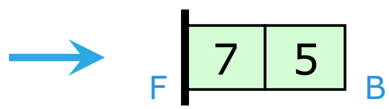
Queues
What a Queue Is — Illustration

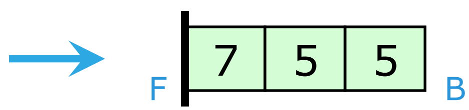
1. Start:
an empty Queue.


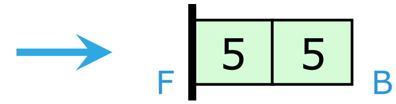
2. Enqueue 2.


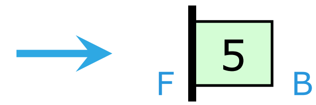
3. Enqueue 7.


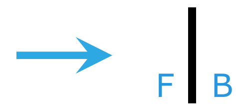
4. Dequeue.


5. Enqueue 5.


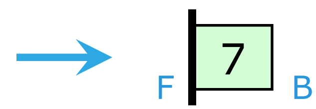
6. Enqueue 5.


7. Dequeue.


8. Dequeue.


9. Dequeue.


Queue is empty again.

10. Enqueue 7.


11. Etc. …

Compare this with Stack!

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 languishes 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

As with a Stack, there is essentially only one good interface to a Queue:

- Data
  - A sequence of data items.
- Operations
  - **getFront**. Look at front item.
  - **enqueue**. Add an item to the back.
  - **dequeue**. Remove front item.
  - To avoid errors we need information about empty state (or size):
    - **isEmpty**. Returns true if queue is empty.
  - Then, of course, we need bookkeeping:
    - **create**.
    - **destroy**.
    - Again, I will add the usual **copy** operations.

Three primary operations.

Queues
Implementation — #1: Sequence Wrapper

As with a Stack, a Queue is often implemented as a wrapper around a Sequence type.

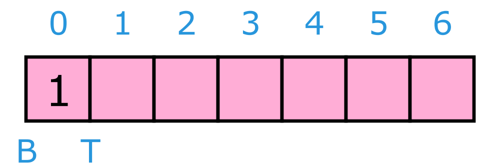
- We would need to use a Sequence type that has fast insertion at one end and fast removal at the other end.
  - NOT a (smart) array.
  - *Maybe* a Singly Linked List …
    - With the right interface. We would need to maintain an iterator to the last element. We can then insert at the end and remove at the beginning. Since we never do remove-at-end, we can always update the iterator when it changes.
  - A Doubly Linked List works.
  - Something like **`std::deque`** works.
- As with a Stack, it is likely that the Queue operations are essentially already implemented.
  - We typically only need to write a bunch of one-line functions.

Queues
Implementation — #2??: Array + Markers

Suppose we try something simpler: put our data in an array with markers indicating the ends.

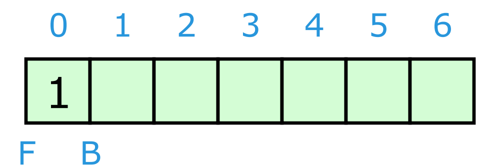
Consider a Stack based on an array with top & bottom markers.

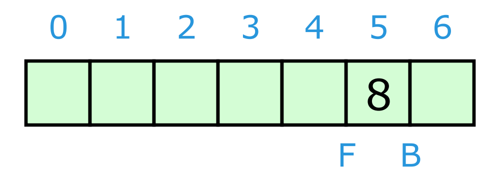
- Begin with a single item on the Stack: a "1" in array element 0.
- Do **`push(8)`** five times, and **`pop()`** five times.
- Result: Exactly the Stack we started with.



Now consider a Queue based on an array with front & back markers.

- Begin with a single item in the Queue: a "1" in array element 0.
- Now do **`enqueue(8)`** five times and **`dequeue()`** five times.
- Result: The single data item in the Queue is an **`8`** in array item 5.
- If the size of the array is as pictured, then two more **`enqueue`** operations will result in the data "crawling" off the end of the array.



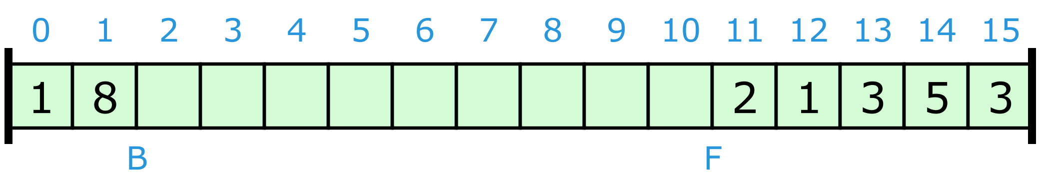


This "crawling data" can make Queues trickier to implement than Stacks.

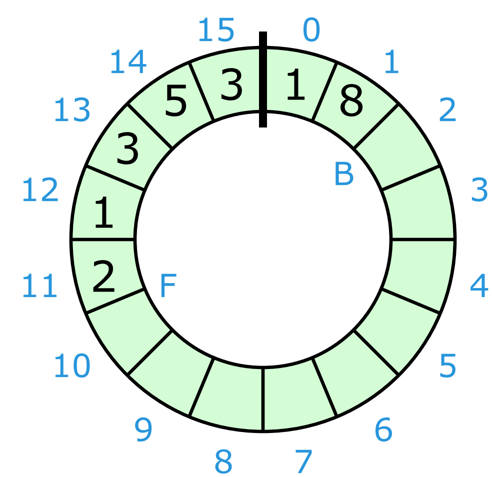
Queues
Implementation — #2: Circular Buffer [1/3]

When we store a Queue in an array with markers, we can deal with "crawling data" using a **circular buffer**.

- A circular buffer is just an ordinary Sequence. However, we think of the ends as being joined.
- We also have markers indicating the front and back of the Queue.
- We generally do not expand or contract the Sequence itself when the Queue expands or contracts; we just move the markers.
- Note that we might still need to expand the Sequence if it fills up.



Physical Structure



Logical Structure

Queues
Implementation — #2: Circular Buffer [2/3]

A circular buffer can be simply an array. We need to know:

- The number of elements in the array.
- The subscript of the front item.
  - When dequeuing, we do `frontsubs = (frontsubs + 1) % array_size`.
- The size of the Queue (that is, the number of items in it).
  - The subscript of the back item is `(frontsubs + queue_size – 1) % array_size`, if `queue_size != 0`.

This is a good way of implementing a Queue that will never exceed some smallish size. For a Queue that can get large:

- We may want to add automatic reallocation.
- This works in much the same way it does for smart arrays.
- When reallocating, be careful to copy items to the right places.

Queues
Implementation — #2: Circular Buffer [3/3]

What is the order of each of the following operations for a Queue implemented using an array-based circular buffer?

- **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`**, in **`<queue>`**.

- Again, STL documentation calls **`std::queue`** a "container adapter", not a "container".

As with **`std::stack`**, **`std::queue`** is a wrapper around a container that you choose.

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

- "**`T`**" is the value type.
- "*container*<**`T`**>" can be **any** standard-conforming container with value type **`T`** and the required member functions (including **`push_back`**, **`pop_front`**, and **`front`**).
- In particular *container* can be **`std::deque`** or **`std::list`**.
  - But not **`std::vector`** or **`std::basic_string`**; these have no **`pop_front`**.

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

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

Queues
In the C++ STL — Notes

Efficiency issues for **`std::queue`** are just like **`std::stack`**.

- Good overall performance is gotten from **`std::deque`**.
- Good worst-case performance is gotten from **`std::list`**, at the expense of memory management overhead.

Functions in **`std::queue`**.

- Enqueue is "**`push`**".
- Dequeue is "**`pop`**".
- GetFront is "**`front`**".
- And comparison operators are defined, etc.

About **`std::deque`**.

- It seems that **`std::deque`** exists primarily to serve as a basis for **`std::queue`** (and **`std::stack`**).
- I have never had occasion to use **`std::deque`** by itself.
    - But maybe you will …

Queues
Applications

Queues are used to mediate **asynchronous communications**.

- *Synchronous* = coordinated in time.
  - Example: I'll call you on the phone at 3 p.m. (we both stop everything at the agreed time and deal with the call).
- *Asynchronous* = not coordinated in time.
  - Example: I send you an e-mail (and you read and answer it when you can).
  - More relevant example: Computer sends document to printer. Printer prints it when it can.

The "waiting in line" behavior of Queues makes asynchronous communication work.

- Sender enqueues a message whenever it has one to send.
- Receiver dequeues a message whenever processing capability is available.
- All messages eventually get processed.

Applications of Queues often involve requests to use some limited resource (an I/O channel, a device, etc.), with requests waiting in line.

- In a print Queue, print jobs wait to be printed.
- In a program with a graphical user interface, user input is often processed in the form of "events". An event might be a mouse click or a keypress. Events will be stored in an event Queue.

Unit Overview
The Basics of Trees

We now begin a unit covering a very different basis for ADTs & data structures: **trees**.

Major Topics

- Introduction to Trees
- Binary Trees
- Binary Search Trees
- Treesort

After this, we look at Tables & Priority Queues.

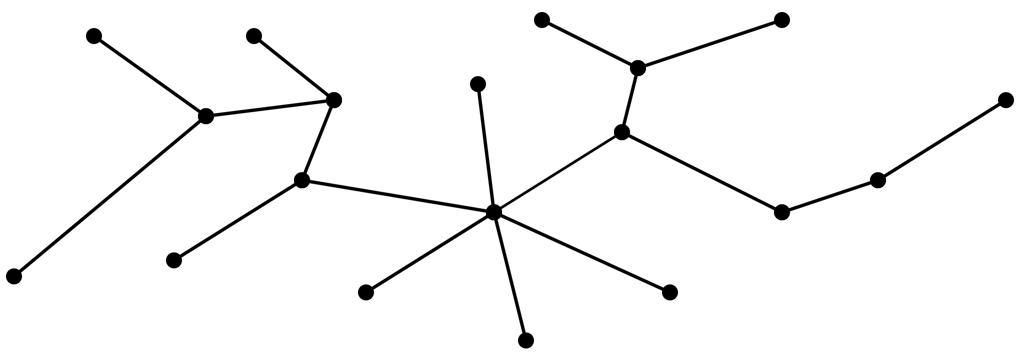
- Some of the more interesting kinds of trees (Binary Heaps, 2-3 Trees, 2-3-4 Trees, Red-Black Trees, AVL Trees) will be covered in this next unit.

Unit Overview
The Basics of Trees

Major Topics

- Introduction to Trees
- Binary Trees
- Binary Search Trees
- Treesort

By a **tree**, mathematicians mean a structure like this:

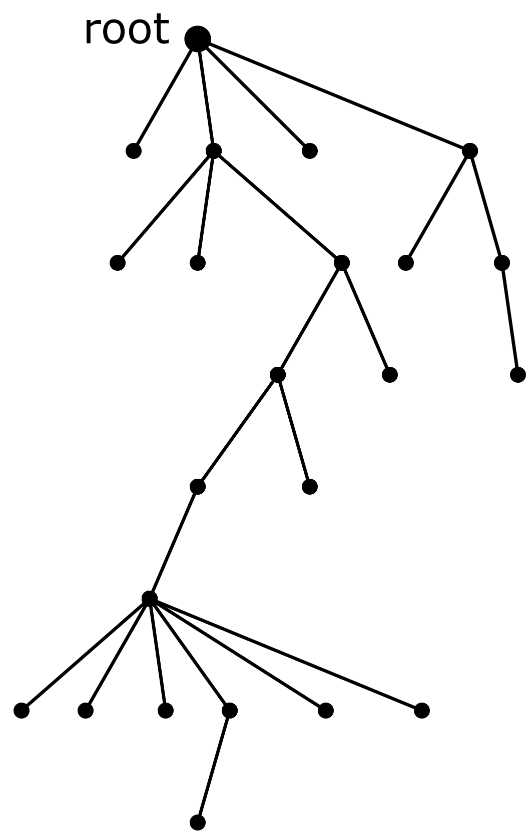


- Each dot is called a **vertex** (note the Latin plural "**vertices**") or a **node**.
  - I will use *vertex* for the element of the tree as a conceptual object, and *node* for the small data substructure representing it.
- Each line is called an **edge**.
- Each edge joins two vertices.
- A tree is connected (all one piece) and there are no cycles.

Often we use trees to represent hierarchical structures.

We place one vertex at the top, and we call it the **root**. Each other vertex of the tree hangs from some vertex. The result is a **rooted tree**.

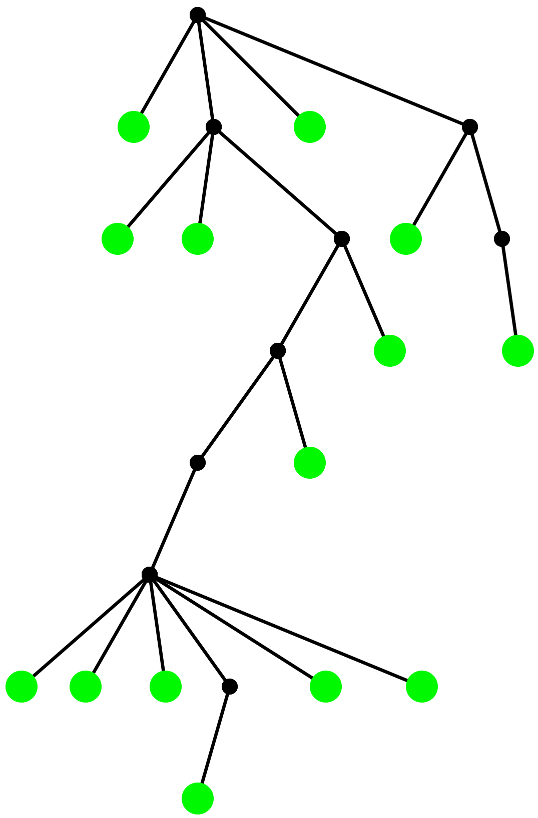
From now on in this class, "tree" means "rooted tree".



Rooted Trees — Terminology [1/5]

Some of the terminology for rooted trees comes from plants.

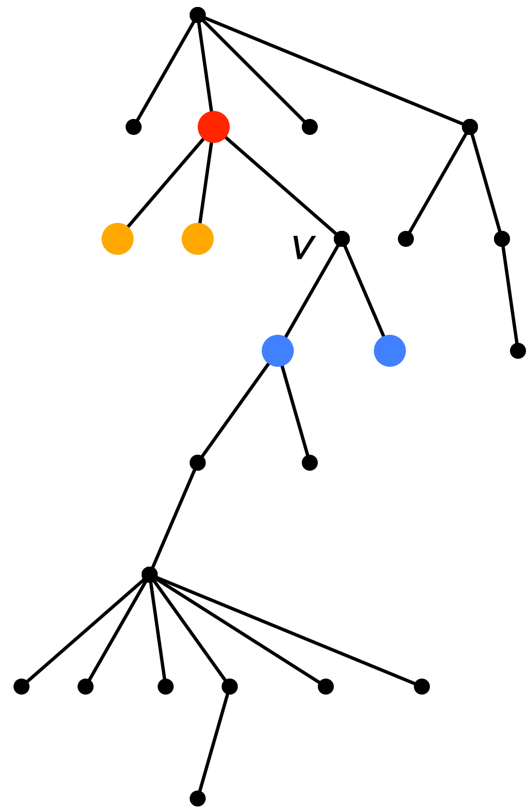
- “Root” is an obvious example.
- Another: A vertex with nothing hanging off of it is called a **leaf**.
  - Leaves are shown in green.
  - What if a tree has just one vertex?



Introduction to Trees
Rooted Trees — Terminology [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 (shown in red) is *v*'s **parent**.
  - Every vertex except the root has exactly one parent.
- The vertices that hang from *v* (shown in blue) are *v*'s **children**.
  - A leaf has no children.
- The other children of *v*'s parent (shown in orange) are *v*'s **siblings**.



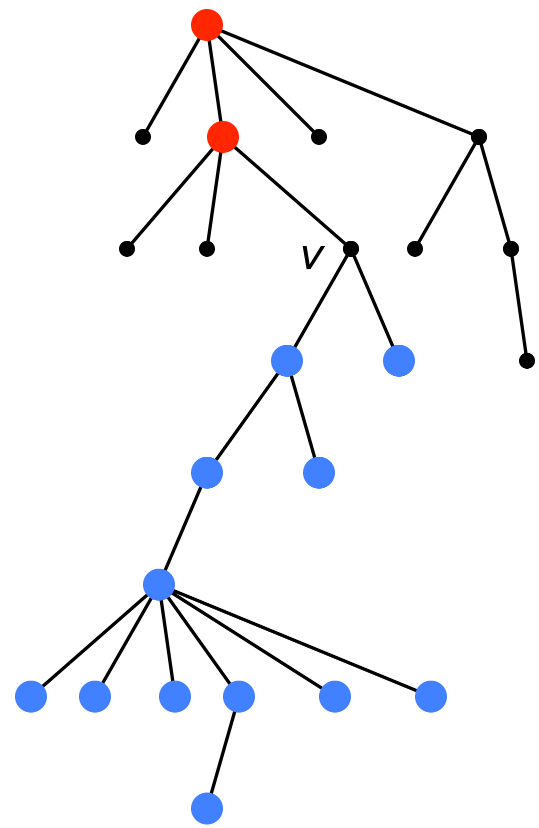
Rooted Trees — Terminology [3/5]

The parent of *v*, and its parent, and its parent, etc., are *v*' s **ancestors**.

- These are shown in red.

v' s children, and their children, and their children, etc., are *v*' s **descendants**.

- These are shown in blue.



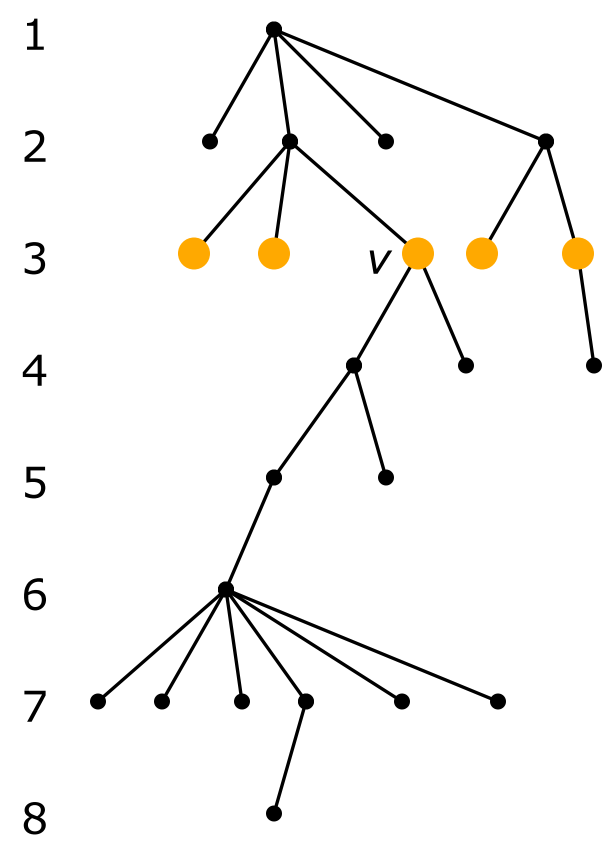
Introduction to Trees
Rooted Trees — Terminology [4/5]

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.
- Level 3, which includes *v*, is shown in orange.
- We often draw vertices at the same level in a horizontal row.

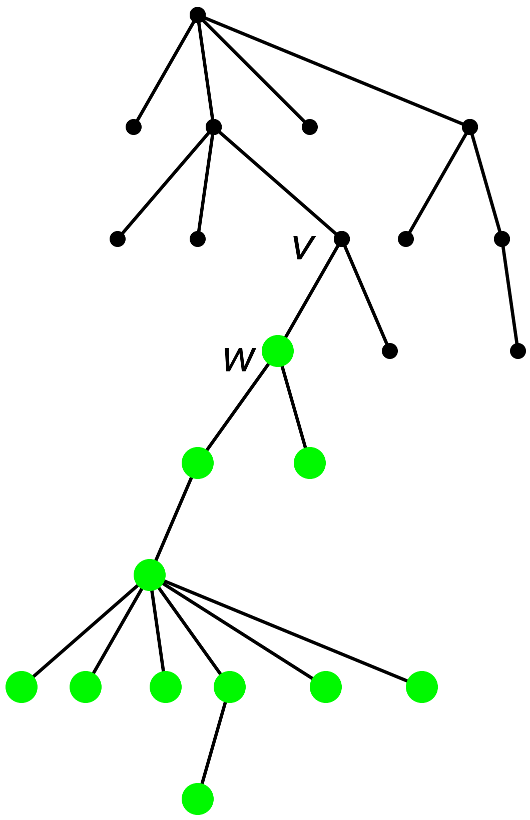
The **height** of a tree is the number of levels.

- This tree has height 8.
- *Note: Some people define "height" slightly differently.*



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*.



A "general tree" is a somewhat more precise version of what we have been talking about.

- A **general tree** consists of a node (called the *root*) and zero or more subtrees of the root, each of which is a general tree.
- Note that a general tree must have at least one node.

The above is a **recursive definition**.

Binary Trees
Overview

Our next ADT is **Binary Tree**.

We will cover:

- What a Binary Tree is.
- Three special kinds.
- Traversals.
- Implementation.
- Applications.

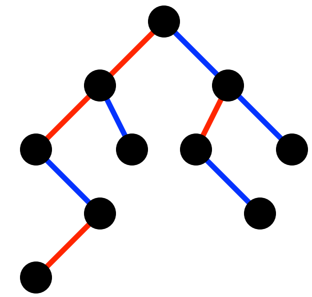
What is missing above?

- "Binary Trees in the C++ STL", because there aren't any.
  - Not in the *interface*, anyway. They are used internally.

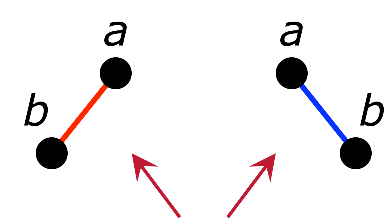
Binary Trees
What a Binary Tree Is — Idea

A **Binary Tree** consists of a set *T* of nodes so that either:

- *T* is empty (no nodes), or
- *T* consists of a node *r*, the root, and two subtrees of *r*, each of which is a Binary Tree:
  - the **left subtree**, and
  - the **right subtree**.



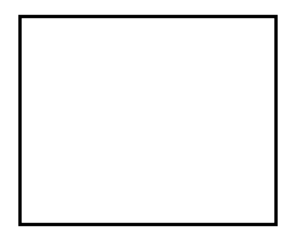
We make a strong distinction between left and right subtrees. Sometimes, we use them for very different things.



Different
Binary Trees

An **empty** Binary Tree is a Binary Tree with no nodes.

Empty Binary Tree



Binary Trees
What a Binary Tree Is — ADT

Data

- A set of nodes.

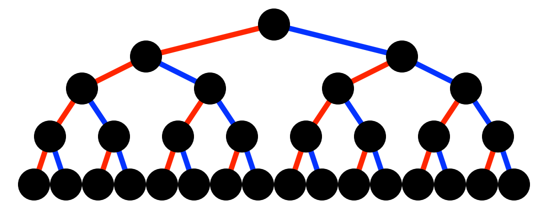
Operations

- **Create** (empty).
- **Create**, given a root and two subtrees.
- **Destroy**.
- **isEmpty**.
- **getRootData** & **setRootData**.
  - Access to data in root node.
- **attachLeft** & **attachRight**.
  - Attach a child to the root.
- **attachLeftSubtree** & **attachRightSubtree**.
  - Attach a subtree to the root.
- **detachLeftSubtree** & **detachRightSubtree**.
  - Detach a subtree from the root.
- **leftSubtree** & **rightSubtree**.
  - Returns a subtree.
- **preorderTraverse**, **inorderTraverse**, & **postorderTraverse**.
  - Visit all nodes in the appropriate order.

Binary Trees
Three Special Kinds

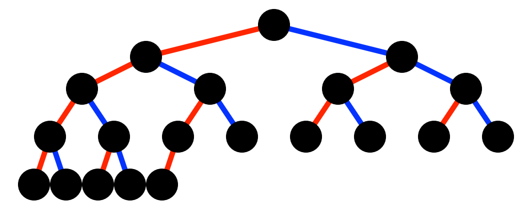
Full Binary Tree

- Leaves are all in the same level.
- All other nodes have two children each.



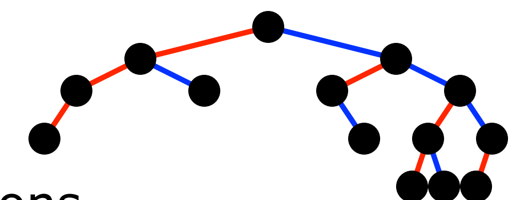
Complete Binary Tree

- All levels above bottom are completely full.
- Bottom level is filled left-to-right.
- Importance: As such trees grow, nodes must be added in a particular order. This gives them a useful array representation, which we look at later.



Balanced Binary Tree

- Left and right subtrees of each node have heights that differ by at most 1.
- Importance: Height is small, even if there are many nodes. This can allow for fast operations.



A full Binary Tree is complete; a complete Binary Tree is balanced.

Full → Complete → Balanced

Binary Trees
Traversals — Idea

One thing we do with Binary Trees is to "traverse" them.

- **Traversing** a tree means visiting each node.

There are three standard traversals of Binary Trees: preorder, inorder, and postorder.

- The name tells us where the root goes: before, in between, after.

Preorder traversal:

- Root.
- Preorder traversal of left subtree.
- Preorder traversal of right subtree.

Inorder traversal:

- Inorder traversal of left subtree.
- Root.
- Inorder traversal of right subtree.

Postorder traversal.

- Postorder traversal of left subtree.
- Postorder traversal of right subtree.
- Root.

Binary Trees
Traversals — Example

Write preorder, inorder, and postorder traversals of the Binary Tree to the right.

Preorder: **1** **2** 4 5 3

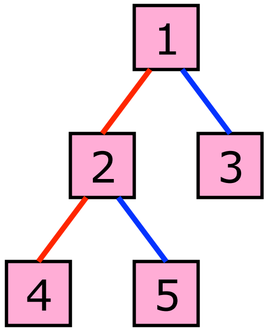
root — left subtree — right subtree

Inorder: 4 **2** 5 **1** 3

left subtree — root — right subtree

Postorder: 4 5 **2** 3 **1**

left subtree — right subtree — root



Binary Trees
Traversals — A Trick

Given a drawing of a Binary Tree, draw a path around it, hitting the left, bottom, and right sides of each node, as shown.

The order in which the path hits the **left** side of each node gives the **preorder** traversal.

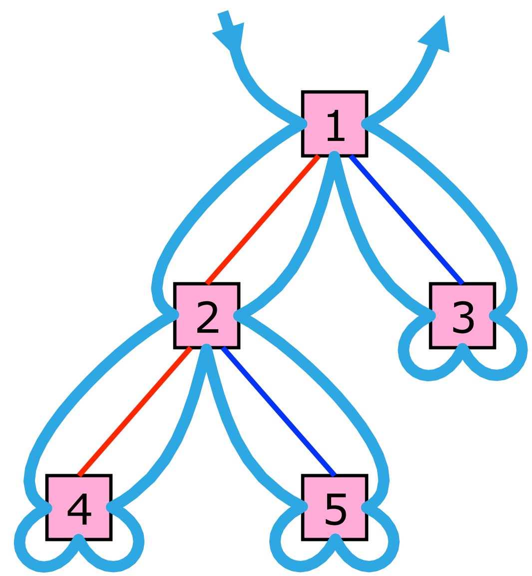
- 1 2 4 5 3

The order in which the path hits the **bottom** side of each node gives the **inorder** traversal.

- 4 2 5 1 3

The order in which the path hits the **right** side of each node gives the **postorder** traversal.

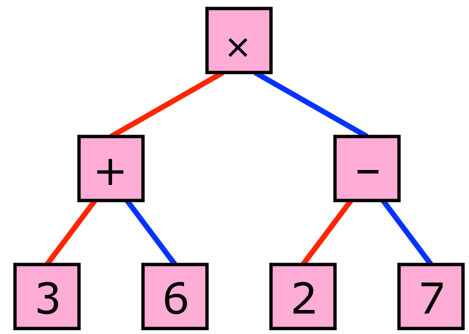
- 4 5 2 3 1



Binary Trees
Traversals — Expressions

Consider the Binary Tree at right.

- This is the **parse tree** of an expression.



Postorder traversal: 3 6 + 2 7 – ×

- This is Reverse Polish Notation for the expression.

Inorder traversal: 3 + 6 × 2 – 7

- This looks like normal infix notation. However, *as an expression*, it is not what we mean; there are problems with precedence.
- Redo: before starting a (sub)tree, insert "(" if there is more than one node in the subtree. Similarly, insert ")" when done.
- Result: ((3 + 6) × (2 – 7)).

Preorder traversal: × + 3 6 – 2 7

- Add parentheses and commas: ×(+(3, 6), –(2, 7)).
- Thinking of "×", "+", and "–" as names of functions, we see that this is standard functional notation.
- This may be clearer: times(plus(3, 6), minus(2, 7)).

Binary Trees
Traversals — Algorithms

There are many reasons why we might traverse a Binary Tree: finding the sum of the data items, printing all data items, etc.

Can we write a single function that can be used to do all these things?

What should our traversal function do? Possibilities:

- It might provide an iterator that goes through the items in the proper order.
- It might return a list holding the data items in the proper order.
- It might be given a function, which it would call for each data item, in order.
  - “**Visitor pattern**”.
  - A restricted version of this might just put each item into an output iterator, like “write” for linked list.

How would we implement the last option above?

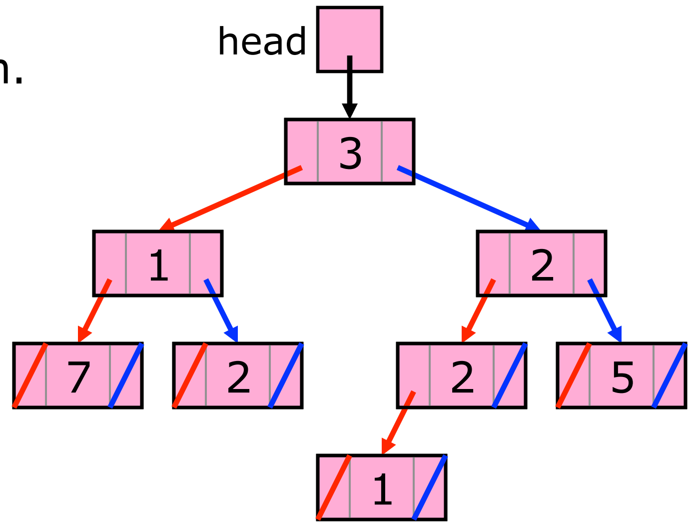
- We could write a recursive function. It would be given a “handle” (pointer? iterator?) to a node and a function to call for each item.
- Algorithm for a preorder traversal:
  - If the handle is null, return.
  - Call the function for the data in the given node.
  - Make a recursive call: left child, given function.
  - Make a recursive call: right child, given function.

For inorder, postorder, move this operation

Binary Trees
Implementation — #1: Pointer-Based [1/2]

A common way to implement a Binary Tree is to use separately allocated nodes referred to by pointers.

- Very similar to our implementation of a Linked List.
- Each node has a data item and two child pointers: left & right.
- Each node owns its subtrees.
  - It is thus responsible for destroying them.
- A pointer is null if there is no child.



Each node *might* also have a pointer to its parent.

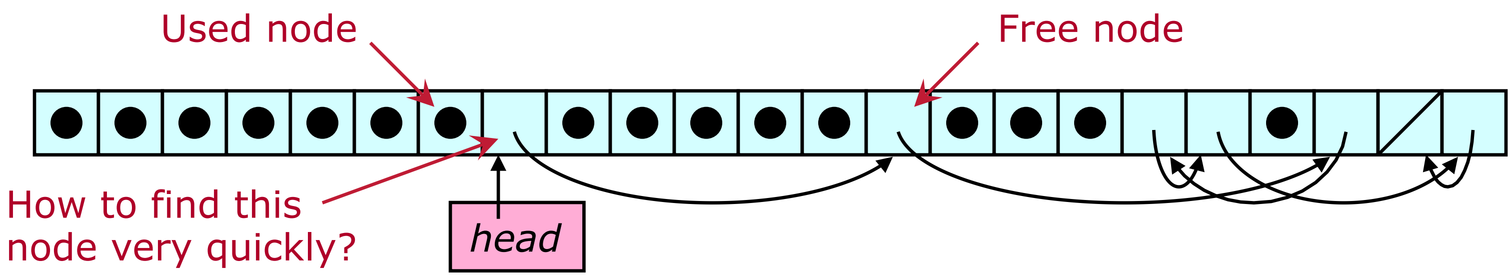
- This would allow some operations to be much quicker.
  - Such as finding the parent of a node.
- Whether we do this, would depend on the purpose of the tree.

Binary Trees
Implementation — #1: Pointer-Based [2/2]

Once again, we *could* put our nodes in an array.

- As before, the primary differences involve memory management: who does it and when it is done.

Q: If we do this, then how can we find a free node quickly?



A: To be able to get new free nodes quickly, we can make free nodes into a Linked List.

Constant time

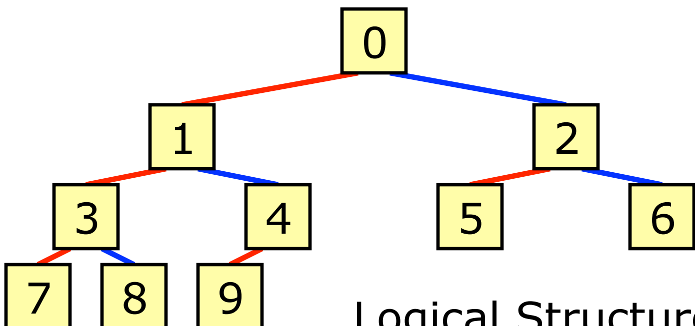
Notes

- This is *easy*. Nodes already have pointers in them (right?). And all we need to do is insert/remove at the beginning of the Linked List.
- This is a common technique, used on all kinds of node-based structures, including Linked Lists.

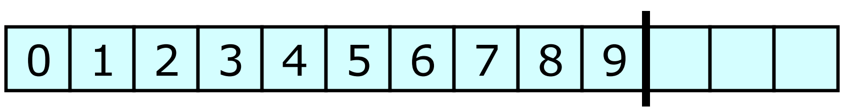
Binary Trees
Implementation — #2: Array-Based Complete [1/2]

A **complete** Binary Tree can be stored efficiently in an array.

- Put the root, if any, at index 0. Other items follow in left-to-right, then top-to-bottom order.
- We need to store *only* an **array** of data items and a record of the number of nodes (***size***).
  - No pointers/indices are required!
- This greatly limits the operations available to us, since we must preserve the property of being complete.



Logical Structure

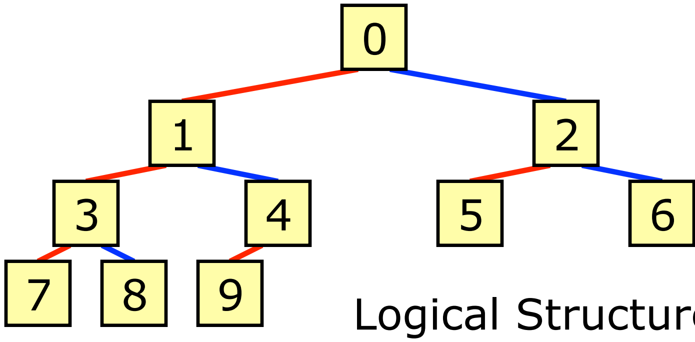


Physical Structure

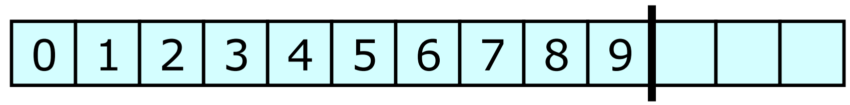
This array-based complete Binary Tree is commonly used to implement a data structure called a "Binary Heap".

- We will discuss this later in the semester.

Binary Trees
Implementation — #2: Array-Based Complete [2/2]



Logical Structure



Physical Structure

Without pointers, how do we move from one node to another?

- The **root**, if any, is at index 0.
  - The root exists if $0 < size$, that is, if the tree is nonempty.
- The **left child** of node k is at index $2k + 1$.
  - The child exists if $2k + 1 < size$.
- The **right child** of node k is at index $2k + 2$.
  - The child exists if $2k + 2 < size$.
- The **parent** of node k is at index $(k – 1)/2$ [integer division].
  - The parent exists if $k > 0$.