Data Structures

ACSL Contest #3

What are data structures and why do they matter?

- “A data structure is a particular way of organizing data in a computer so that it can be used effectively. For example, we can store a list of items having the same data-type using the array data structure.”
  - https://www.geeksforgeeks.org/data-structures/
- “At the heart of virtually every computer program are its algorithms and its data structures. It is hard to separate these two items, for data structures are meaningless without algorithms to create and manipulate them, and algorithms are usually trivial unless there are data structures on which to operate. The bigger the data sets, the more important data structures are in various algorithms.”

Scope of Data Structures for ACSL

- This category concentrates on four of the most basic structures:
  - **Stacks**: usually used to save information that will need to be processed later. Items are processed in a “last-in, first-out” (LIFO) order.
  - **Queues**: Usually used to process items in the order in which requests are generated - a new item is not processed until all items currently on the queue are processed. This is known as a “first-in, first-out” (FIFO) order.
  - **Binary search trees**: used when one is storing items and needs to be able to efficiently process the operations of insertion, deletion, and query (i.e. find out if a particular item is found in the list of items and if not, which item is close to the item in question).
  - **Priority queues**: used like a binary search tree, except one cannot delete an arbitrary item, nor can one make an arbitrary query - one can only find out or delete the smallest element of hte list.
- Questions will cover these data structures and implicit algorithms, not specific to implementation language details.

Stacks

- Supports two operations: PUSH and POP.
- A command of the form PUSH("A") puts the key "A" at the top of the stack.
- The command “X = POP()” removes the top item from the stack and stores its value into variable X.
    - If the stack was empty (because nothing had ever been pushed on it, or if all elements have been popped off of it), then X is given the special value of NIL.
- An analogy to this is a stack of books on a desk: a new book is placed on the top of the stack (pushed) and a book is removed from the top also (popped).
- Some textbooks call this data structure a “push-down stack” or a “LIFO stack”.

Queues

- Operate just like stacks, except items are removed from the bottom instead of the top.
- A command of the form PUSH("A") puts the key "A" at the top of the queue
- The command “X = POP()” removes the item from the bottom of the queue and stores its value into variable X.
    - If the queue was empty (because nothing had ever been pushed on it, or if all elements have been popped off of it), then X is given the special value of NIL.
- A good physical analogy of this is the way a train conductor or newspaper boy uses a coin machine to give change: new coins are added to the tops of the piles, and change is given from the bottom of each.
    - Sometimes the top and bottom of a queue are referred to as the rear and the front respectively. Therefore, items are pushed/enqueued at the rear of the queue and popped/dequeued at the front of the queue. Some textbooks refer to this data structure as a “FIFO stack”.

Stacks VS Queues

- If these operations are applied to a stack, then the values of the pops are: E, R, I, M, A and NIL. After all of the operations, there are three items still on the stack: the N is at the top (it will be the next to be popped, if nothing else is pushed before the pop command), and C is at the bottom.
- If, instead of using a stack we used a queue, then the values popped would be: A, M, E, R, I and NIL. There would be three items still on the queue: N at the top and C on the bottom. Since items are removed from the bottom of a queue, C would be the next item to be popped regardless of any additional pushes.

```
PUSH("A")
PUSH("M")
PUSH("E")
X = POP()
PUSH("R")
X = POP()
PUSH("I")
X = POP()
X = POP()
X = POP()
X = POP()
PUSH("C")
PUSH("A")
PUSH("N")
```

ACSL Specific Note About Pre-2018 Syntax

ACSL Contests pre-2018 used a slightly different, and more liberal, syntax. Consider the following sequence on a stack:

```
PUSH(A)
PUSH(B)
PUSH(C)
POP(X)
POP(Z)
```

After the 5 operations, the stack would be left with A as the only element on the stack, the value of C in variable X, and the value of B in variable Z.

Were the above operations applied to a queue, the queue would be left with C; variable X would contain A; and variable Z would contain B.

Optional Viewing

Reviews what stacks and queues are and some sample code: Data Structures: Stacks and Queues (HackerRank)

Suggested Viewing Ahead of Following Slides

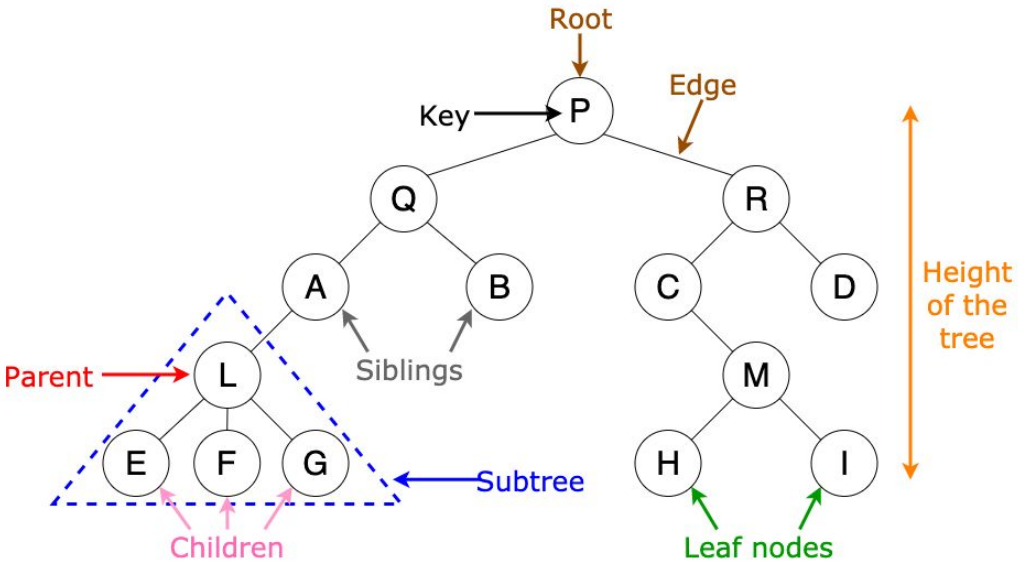
- ACSL assumes you're already acquainted with binary search: this short video eases you into it with a real life analogy: https://www.youtube.com/watch?v=P3YID7liBug
  - You can skip the programming bit while just learning the theory, but I'd suggest returning to it later.

Required Viewing Ahead of Following Slides

- Links binary search to Trees (it's 4 mins without the code and 10 mins with): https://www.youtube.com/watch?v=oSWTXtMglKE
  - But ACSL does provide pseudo-code you should be acquainted with, so watching the programming bit would be helpful.

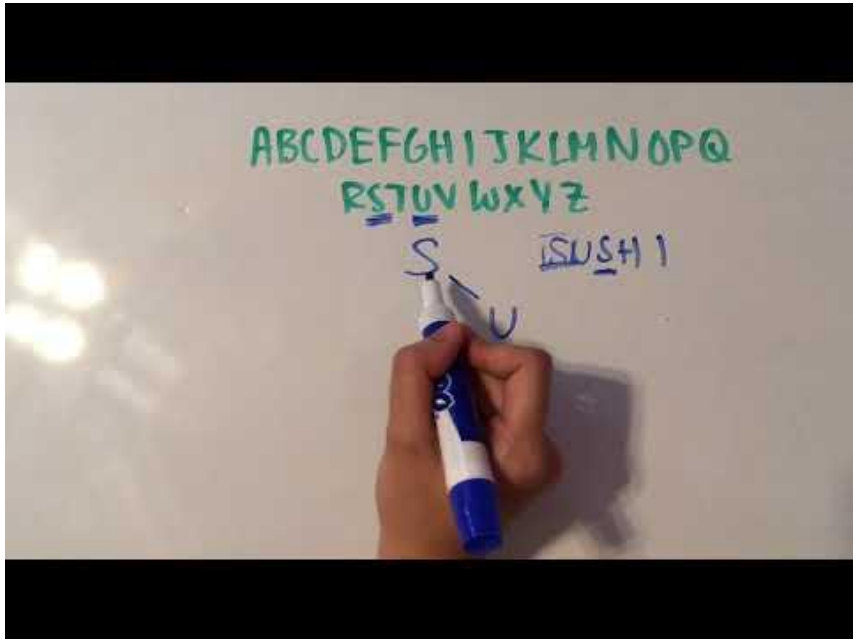
Trees

- *Trees*, in general, use the following terminology: the *root* is the top node in the tree; *children* are the nodes that are immediately below a *parent* node; *leaves* are the bottom-most nodes on every branch of the tree; and *siblings* are nodes that have the same immediate parent.



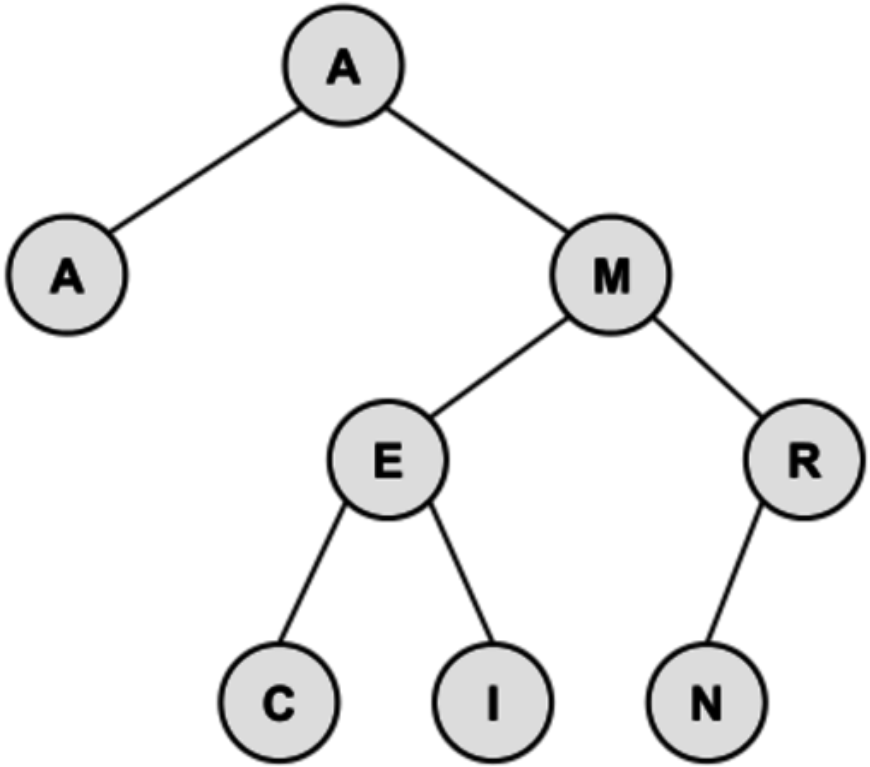
Constructing a Binary Search Tree

- A *binary search tree* is composed of nodes having three parts:
  - Information (or a key)
  - A pointer to a left child
  - A pointer to a right child.
- It has the property that the key at every node is always greater than or equal to the key of its left child, and less than the key of its right child.
- Watch this 2 minute video to see how binary search trees are constructed:



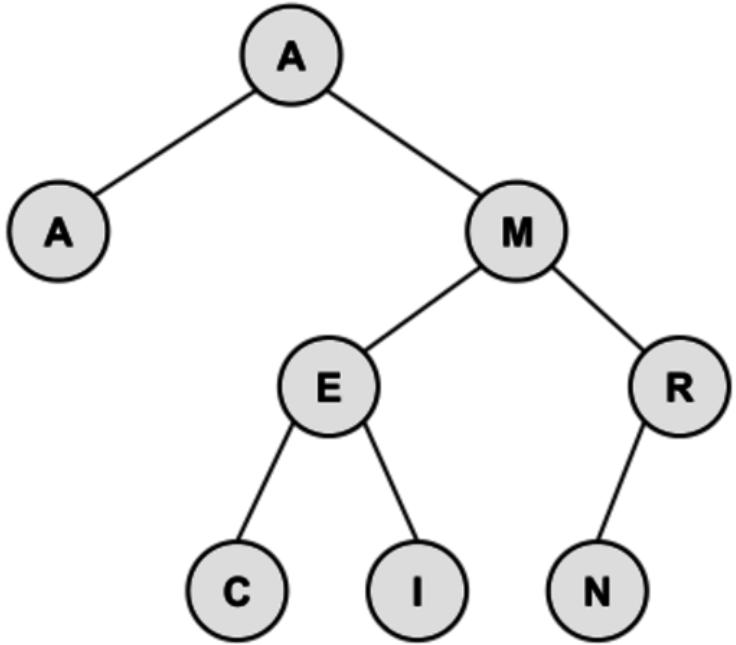
Constructing a Binary Search Tree

- The following tree is built from the keys A, M, E, R, I, C, A, N in that order:



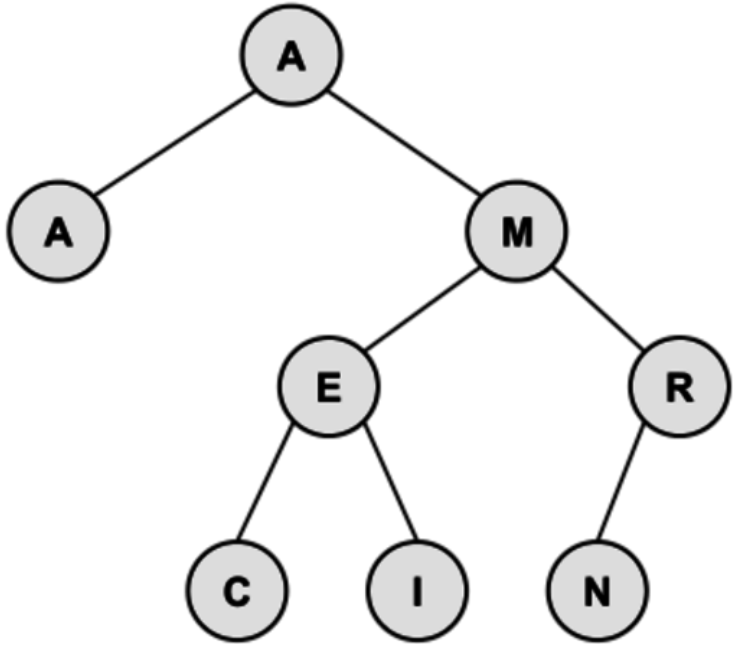
Binary Search Tree Structure

- The *root* of the resulting tree is the node containing the key A.
- ACSL convention places duplicate keys into the tree as if they were less than their equal key.
    - (In some textbooks and software libraries, duplicate keys may be considered larger than their equal key.)
- The tree has a *depth* (sometimes called height) of 3 because the deepest node is 3 nodes below the root.
- The root node has a depth of 0.



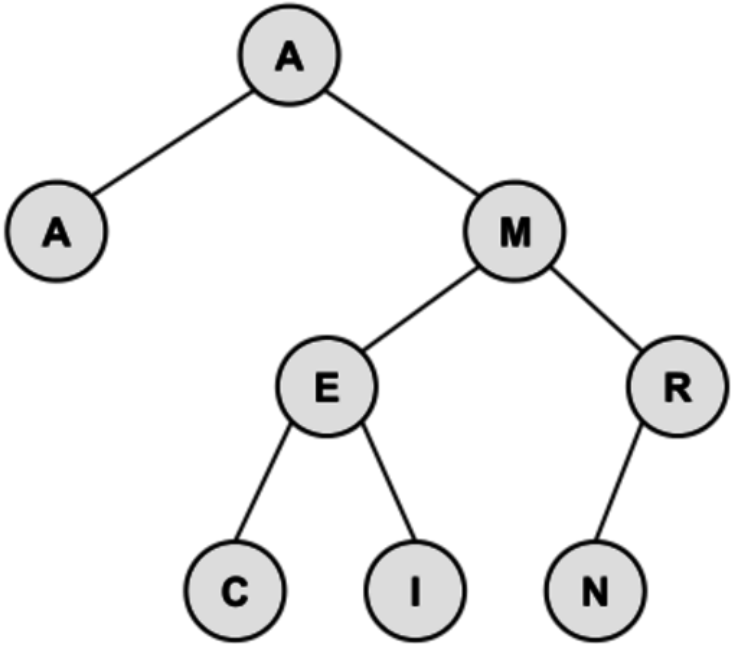
Binary Search Tree Structure

- Nodes with no children are called leaf nodes. There are four of them in the tree:
  - A
  - C
  - I
  - N
- ACSL convention is that an *external node* is the name given to a place where a new node could be attached to the tree.
  - (In some textbooks, an external node is synonymous with a leaf node.)
- In the final tree above, there are 9 external nodes; these are not drawn.



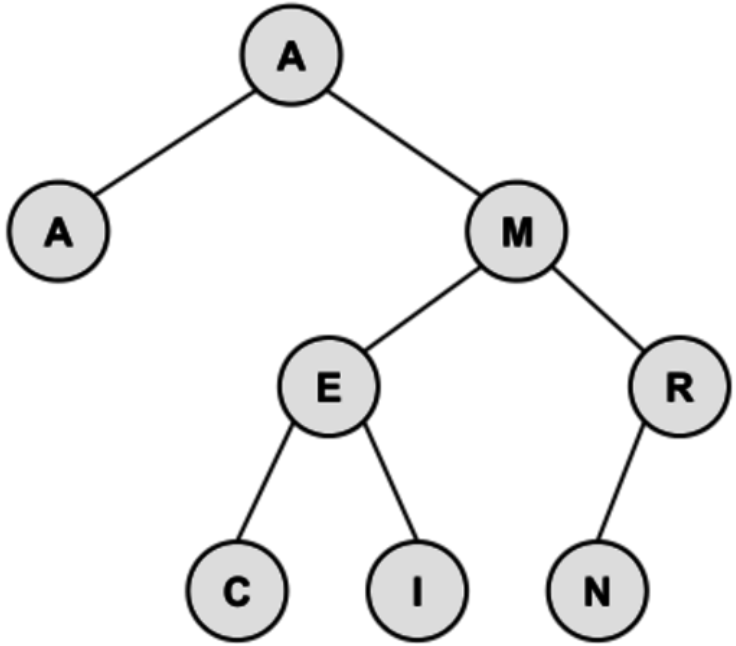
Binary Search Tree Structure

- The tree has an *internal path length* of 15 which is the sum of the depths of all nodes.
- It has an *external path length* of 31 which is the sum of the depths of all external nodes.
- This video goes through an example of solving for the internal path length: Binary Search Tree ACSL Problem (Internal Path Length) (Tangerine Code)
    - This is optional as we will go through a few examples together.



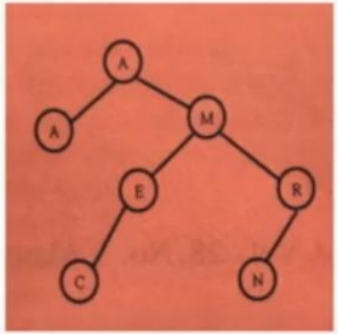
Binary Search Trees Operations: Insert

- A binary search tree can support the operations insert, delete, and search.
- Insert:
  - To insert the N (the last key inserted), 3 *comparisons* were needed against the root A (>), the M (>), and the R (≤).



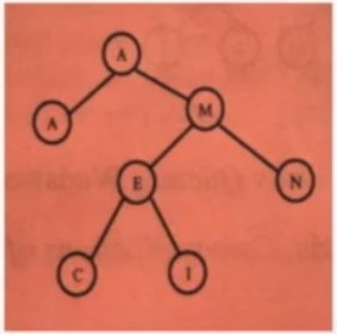
Binary Search Trees Operations: Delete

- Delete:



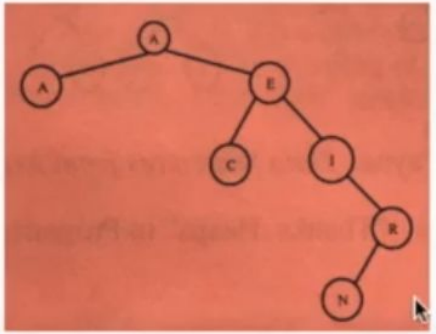
Delete 'I'

I has no nodes, simply remove the item



Delete 'R'

R has one node, replace R with its previous node (N)



Delete 'M'

Father: A Left Child of M: E<CI
Right Child of M: R-N

Delete M, Put LC where M was, stick RC onto LC

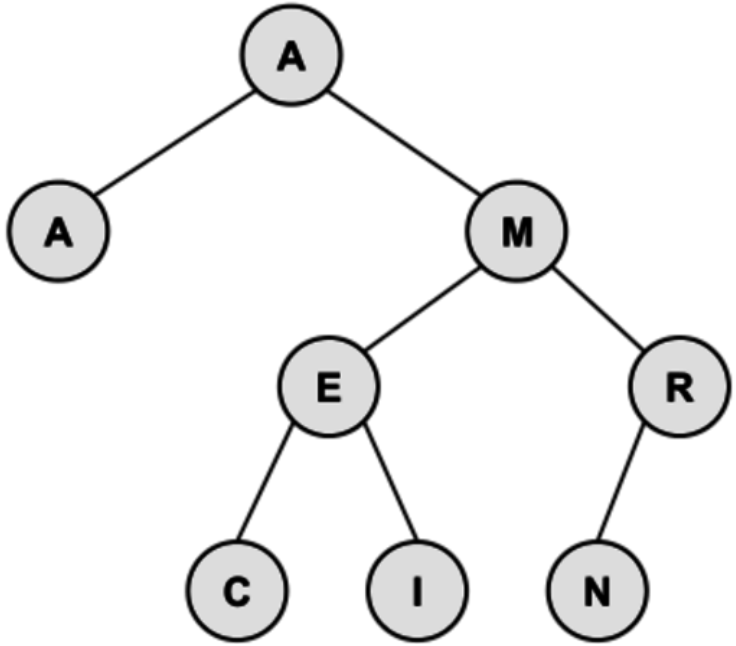
Binary Search Tree Deletion Pseudo-Code

Deleting from a binary search tree is a bit more complicated. The algorithm we'll use is as follows:

```
p = node to delete
f = father of p
if (p has no children)
  delete p
else if (p has one child)
  make p's child become f's child
  delete p
else if (p has two children)
  l = p's left child (it might also have children)
  r = p's right child (it might also have children)
  make l become f's child instead of p
  stick r onto the l tree
  delete p
end if
```

Binary Search Trees Operations: Search

- Search:
  - We start at the root and the compare the value to be searched with the value of the root.
  - If it's equal we are done with the search.
  - If it's less than the node's value we look to the left of the subtree, because in a binary search tree all elements to the left of a node are less than the node and all elements to the right are greater than.
  - Searching an element in the binary search tree is basically this traversal in which at each step we will go either towards left or right, consequently discarding one of the sub trees at each step.
- Pick a letter from the word AMERICAN and try finding it via the Search operation.



Searching a Binary Tree Pseudo-Code

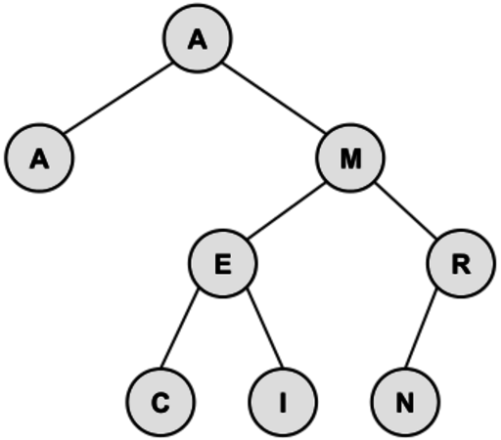
To search for a node in a binary tree, the following algorithm (in pseudo-code) is used:

```
p = root
found = FALSE
while (p ≠ NIL) and (not found)
  if (x < p's key)
    p = p's left child
  else if (x > p's key)
    p = p's right child
  else
    found = TRUE
  end if
end while
```

Here's some actual code: https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

Binary Search Tree: Traversals

1. Inorder Traversal: To perform an *inorder* traversal of the tree, recursively traverse the tree by first visiting the left child, then the root, then the right child.
    a. In the tree to the right, the nodes are visited in the following order: A, A, C, E, I, M, N, and R.
2. Preorder Traversal: A *preorder* traversal (root, left, right) visits in the following order: A, A, M, E, C, I, R, and N.
3. Postorder Traversal: A *postorder* traversal (left, right, root) is: A, C, I, E, N, R, M, A.
    a. Inorder traversals are typically used to list the contents of the tree in sorted order.



Binary Search Trees: Good to Know

- Binary search trees handles operations efficiently for *balanced* trees.
  - In a tree with 1 million items, one can search for a particular value in about $\log_2 1{,}000{,}000 \approx 20$ steps. Items can be inserted or deleted in about as many steps, too.
- However, consider the binary search tree resulting from inserting the keys A, E, I, O, U, Y which places all of the other letters on the right side of the root "A". This is very unbalanced; therefore, sophisticated algorithms can be used to maintain balanced trees.
- Binary search trees are “dynamic” data structures that can support many operations in any order and introduces or removes nodes as needed.
- There are also general trees that use the same terminology, but they have 0 or more *subnodes* which can be accessed with an array or linked list of pointers. *Pre-order* and *post-order* traversals are possible with these trees, but the other algorithms do not work in the same way. Applications of general trees include game theory, organizational charts, family trees, etc.
- *Balanced* trees minimize searching time when every leaf node has a depth of within 1 of every other leaf node. *Complete* trees are filled in at every level and are always balanced. *Strictly binary* trees ensure that every node has either 0 or 2 subnodes.

Priority Queues (You're almost through!)

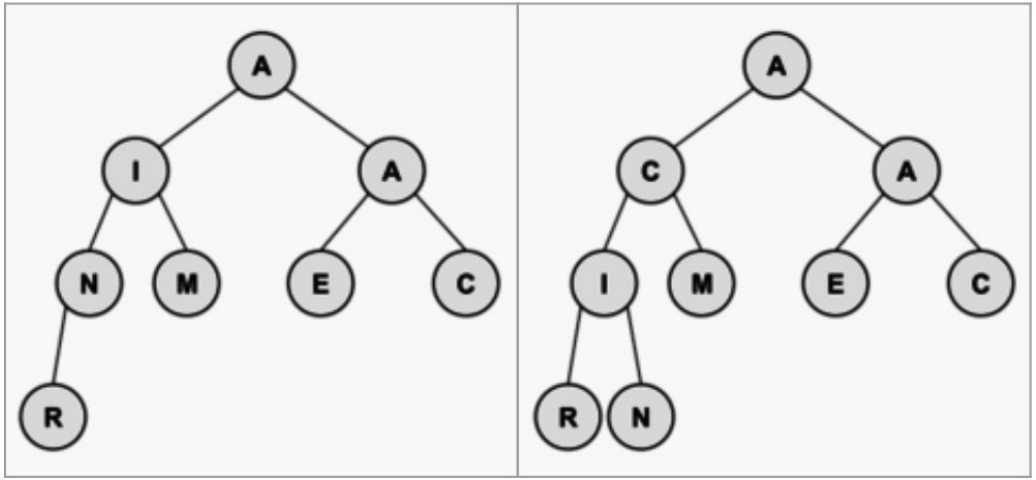
- A priority queue is quite similar to a binary search tree, but one can only delete the smallest item and retrieve the smallest item only.
- These insert and delete operations can be done in a guaranteed time proportional to the log (base 2) of the number of items; the retrieve-the-smallest can be done in constant time.
- The standard way to implement a priority queue is using a *heap* data structure. A heap uses a binary tree (that is, a tree with two children) and maintains the following two properties:
    - Every node is less than or equal to both of its two children (nothing is said about the relative magnitude of the two children), and the resulting tree contains no "holes".
    - That is, all levels of the tree are completely filled, except the bottom level, which is filled in from the left to the right.

Suggested Viewing Ahead of Following Slides

- First 3 minutes are theory, the rest is code:Data Structures: Heaps (HackerRank)
    - Same recommendations as before apply
- When the smallest item is at the root of the heap, the heap is called a *min-heap*. Of course, A *max-heap* is also possible and is common in practice. An efficient implementation of a heap uses an array that can be understood conceptually by using a tree data structure.

Priority Queue Insertion

The algorithm for insertion is not too difficult: put the new node at the bottom of the tree and then go up the tree, making exchanges with its parent, until the tree is valid. The heap at the left was building from the letters A, M, E, R, I, C, A, N (in that order); the heap at the right is after a C has been added.



Priority Queue Deletion

The smallest value is always the root. To delete it (and one can only delete the smallest value), one replaces it with the bottom-most and right-most element, and then walks down the tree making exchanges with the smaller child in order to ensure that the tree is valid. The following pseudo-code formalizes this notion:

```
b = bottom-most and right-most element
p = root of tree
p's key = b's key
delete b
while (p is larger than either child)
  exchange p with smaller child
  p = smaller child
end while
```

Further Resources

- Binary Search Tree challenge on HackerRank: https://www.hackerrank.com/challenges/30-binary-search-trees/problem
- Khan Academy article on Binary Search: https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search
- ACSL's wiki page has a number of suggested videos if you'd like to learn more: http://www.categories.acsl.org/wiki/index.php?title=Data_Structures

Sample Problem 1

Consider an initially empty stack. After the following operations are performed, what is the value of Z?

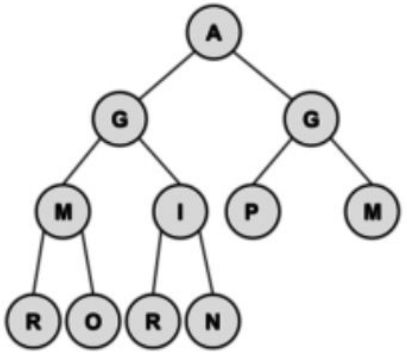
```
PUSH(3)
PUSH(6)
PUSH(8)
Y = POP()
X = POP()
PUSH(X-Y)
Z = POP()
```

Solution: the first POP stores 8 in Y. The second POP stores 6 in X. Then, 6-8 = -2 is pushed onto the stack. Finally, the last POP removes the -2 and stores it in Z.

Sample Problem 2

Create a min-heap with the letters in the word PROGRAMMING. What are the letters in the bottom-most row, from left to right?

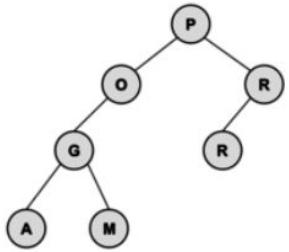
Solution: The bottom row contains the letters RORN, from left-to-right. Here is the entire heap:



Sample Problem 3

Create a binary search tree from the letters in the word PROGRAM. What is the internal path length?

Solution: When drawing the tree, P has a depth of 0, O and R have a depth of 1, G and R have a depth of 2, and A and M have a depth of 3. Therefore, the internal path length is 2*1 + 2*2 + 2*3 = 12. Here is the tree:



Past Contests: Senior 1

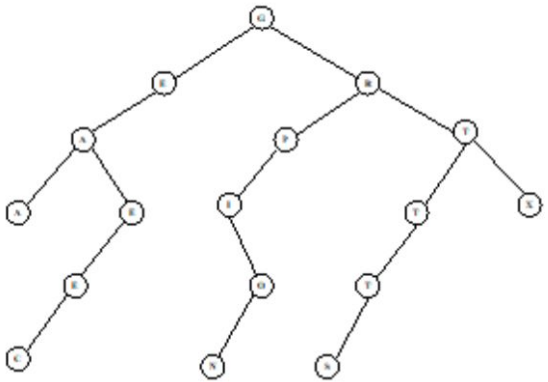
3. Data Structures

List all nodes that have two children in the binary search tree for:

GREATEXPECTATIONS

3. Data Structures

The binary tree formed is shown on the right. The nodes with 2 children are: G, A, R, and T.



3. G, A, R, T

Past Contests: Senior 2

4. Data Structures

The operation SW switches a queue to a stack and a stack to a queue. Given an initially empty queue and the following sequence of operations, what would be the next POPPED element?

PUSH(M), PUSH(I), PUSH(S), POP(X), PUSH(S), PUSH(H), POP(X), POP(X), PUSH(A), POP(X), SW, PUSH(V), PUSH(I), POP(X), PUSH(S), PUSH(H), PUSH(A), POP(X), PUSH(M), POP(X), POP(X)

4. Data Structures

The queue is constructed using FIFO as follows: M, MI, MIS, IS, ISS, ISSH, SSH, SH, SHA, HA. Switch to a stack.

The stack is constructed using LIFO as follows: HAV, HAVI, HAV, HAVS, HAVSH, HAVSHA, HAVSH, HAVSHM, HAVSH, HAVS.

The next element popped would be S.

4. S