```
// Heaps:

// MIN HEAP:
// - The root would be the smallest element and then the nodes
// would get larger and larger as you move down the tree with the
// largest node being the left child.

// MAX HEAP:
// - The root would be the largest element and then the nodes
// would get smaller and smaller as you move down the tree with the
// smallest node being the left child.

// Insertion: Inserted element always goes to the next empty
// spot looking from Top -> Bottom and then Left -> Right, and
// then to put it in the right position you need to bubble up
// by compares the current node with parent node.

// Removing: You empty the root and then replace the root with
// the last element then bubble it down by comparing it to the
// children of that parent.

// Implementation: No gaps in the heap, and we can use an array to
// store these values bc they are in a specfic spot. This makes them very
// compact, and a simple equation can map from an index to its left
// child, its right child or its parent.

// Code:

class MinIntHeap{
    capacity = 10;
    size = 0;
    int[] items = new int[capacity];

    int getLeftChildIndex(int parentindex) { return 2 * parentindex + 1 }
    int getRightChildIndex(int parentindex) { return 2 * parentindex + 2 }
    int getParentIndex(int childindex) { return ( childindex - 1 ) / 2 }

    bool hasLeftChild(int index) { return getLeftChildIndex(index) < size}
    bool hasRightChild(int index) { return getRightChildIndex(index) < size}
    bool hasParent(int index) { return getParentIndex(index) >= 0 }

    int leftChild(int index) { return items[getLeftChildInde(index)]}
    int rightChild(int index) { return items[getRightChildIndex(index)]}
    int parent(int index) { return items[getParentIndex(index)]}

    void swap(int index1, int index2){
    // swap values at these two indicies
        int temp = items[index1];
        items[index1] = items[index2];
        items[index2] = temp;
    }

    void ensureExtraCapacity() {
        if (size == capacity) {
            items = Arrays.copyOf(items, capacity * 2);
            capacity *= 2;
    // This is doubling the array and then copying the array into
    // the new array that is double the capacity of the old array.
    // O(logn) and this is better than just constantly adding
    // another space because that would be O(n^2) aka not as efficient.
        }
    }

    int peak() {
    // returns 1st element in the array (root) if not empty.
        if (size == 0) throw new IllegalException();
        return items[0];
    }

    int poll() {
    // extracts the min. element and removes it from the array if not empty.
    // So take the last element in the array, move it to the element before it,
    // shrink the size, then heapifyDown()
        if (size == 0) throw new IllegalException();
        int item = items[0];
        items[0] = items[size - 1];
        size --;
        heapifyDown();
        return item;
    }

    void add(int item) {
    // ensure extra capacity, add element into last spot then increase the size,
    // then heapifyUp()
        ensureExtraCapacity();
        items[size] = item;
        size ++;
        heapifyUp();
    }

    void heapifyUp() {
    // start with the last added element then as long as there is a
    // parent item, bubble up the tree while it is out of order
    // then swap the parent with the index.
        int index = size - 1;
        while (hasParent(index) && parent(index) > items[index]) {
            swap(getParentIndex(index), index);
            index = getParentindex(index);
        }
    }
    void heapifyDown() {
    // start at the root and as long as you have a left child, then you have right
    // and as long as it is out of order for any of the nodes, then swap. So this
    // is the step that re-looks as the subtrees to ensure all is heapified.
        int index = 0;
        while (hasLeftChild(index)) {
            int smallerChildIndex = getLeftChildIndex(index);
            if (hasRightChild(index) && rightChild(index) < leftChild(index)) {
                smallerChildIndex = getRightChildIndex(index);
            }
            if (items[index] < items[smallerChildIndex]) {
                break; // done, all is heapified!
            }
            else {
                swap (index, smallerChildIndex);
            }
            index = smallerChildIndex;
        }
    }
}
```

```
from sys import argv

class Node(object):

    def __init__(self, data = None, next = None):
        self.__data = data
        self.__next = next

    def setData(self, data):
        # Set the "data" data field to the corresponding input
        self.__data = data;

    def setNext(self, next):
        # Set the "next" data field to the corresponding input
        self.__next = next

    def getData(self):
        # Return the "data" data field
        return self.__data

    def getNext(self):
        # return the "next" data field
        return self.__next
```

```
class Queue(object):

    def __init__(self):
        self.__head = None  # Initializing to be None for both the head and tail
        self.__tail = None


    def enqueue(self, newData): # use the set functions from node
        # Create a new node whose data is newData and whose next node is null
        # Update head and tail
        # Hint: Think about what's different for the first node added to the Queue

        current_node = Node(newData, None) # Creating a node with neeData, and None as next data

        if self.__tail == None:
            self.__head = current_node # updating the head and tail to be the node if there is
            # nothing for the last position aka empty
            self.__tail = current_node
        else:
            self.__tail.setNext(current_node) # if there is something in the queue, then the tail
            # is the next piece of data, so this is setting that, then updating it.
            self.__tail = self.__tail.getNext()
```

```
    def dequeue(self):
        if self.__head is None: # checking if it is empty
            return None # returning none if the queue is empty
        else:
            temp = self.__head.getData() # temporary node with the data we are wanting to currently return
            self.__head = self.__head.getNext() # the updating the head of the node to be next
            return temp # returning the data for the last data item in the queue currently
```

```
class Stack(object):

    def __init__(self):
        # We want to initialize our Stack to be empty
        self.__head = None  # The top/head is None (null) starting out.

    def push(self, newData):
        current_node = Node(newData, self.__head)  # Making a new node.

        if self.__head is None:
            self.__head = current_node

        else:
            current_node.setNext(self.__head)
            self.__head = current_node
```

```
    def pop(self):
        if self.__head is None:
            return None
        else:
            temp = self.__head.getData()
            self.__head = self.__head.getNext()
            return temp
```

TREE-SEARCH(x, k)

```
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

TREE-SUCCESSOR(x)

```
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

TREE-INSERT(T, z)

```
1   y = NIL
2   x = T.root
3   while x ≠ NIL
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

TRANSPLANT(T, u, v)

```
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

TREE-DELETE(T, z)

```
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)
5  else y = TREE-MINIMUM(z.right)
6      if y.p ≠ z
7          TRANSPLANT(T, y, y.right)
8          y.right = z.right
9          y.right.p = y
10     TRANSPLANT(T, z, y)
11     y.left = z.left
12     y.left.p = y
```

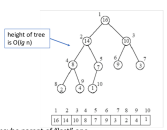
```
push(x):
    s = Q.size()
    Q.enqueue(x) // x is at the tail of the queue now
    for i = 1 to s:
        Q.enqueue(Q.dequeue())
    // Now x is at the head

pop():
    return Q.dequeue()
```

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$
MINIMUM	$\Theta(1)$	$O(\lg n)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$

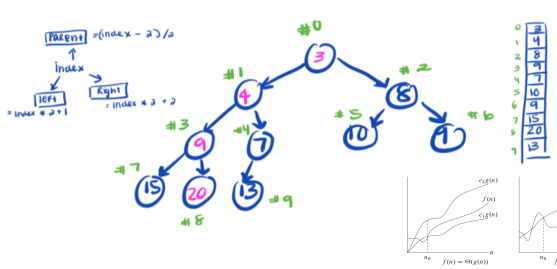
```
preorderTrav(node p)

  print p.value
  if p.left != null
      preorderTrav(p.left)
  if p.right != null
      preorderTrav(p.right)

implement that traversal with a stack:

stack S of node

S.push(T.root)

while (not S.isEmpty)
    p = S.pop
    print p.value
    if p.right!=null
        S.push(p.right)
    if p.left!=null
        S.push(p.left)

implement that traversal with a queue:

queue Q of node

Q.enqueue(T.root)

while (not Q.isEmpty)
    p = Q.dequeue
    print p.value
    if p.right!=null
        Q.enqueue(p.right)
    if p.left!=null
        Q.enqueue(p.left)
```

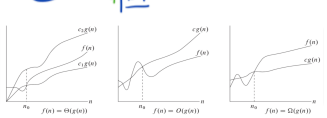
binary heap

- stored in array
- item located in position i
  - parent in location $\lfloor i/2 \rfloor$
  - left child in position $2i$
  - right child in position $2i + 1$
- tree is complete
  - all nodes have two children, except maybe parent of "last" one
- tree maintains heap property
  - value stored at location i is greater than or equal to values stored in both its children
- fact: a binary heap with n elements has the height of $\lfloor \lg n \rfloor$ (why?)



	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ	Yes	Yes	No	No	No
b.	n^k	c^n	Yes	Yes	No	No	No
c.	$\sqrt{n}$	$n^{\sin x}$	No	No	No	No	No
d.	2^n	$2^{n/2}$	No	No	Yes	Yes	No
e.	$n^{\lg c}$	$c^{\lg n}$	Yes	No	Yes	No	Yes
f.	$\lg(n!)$	$\lg(n^n)$	Yes	No	Yes	No	Yes



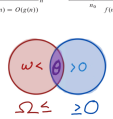


Complexity Performance:

Big Ω ≤ Θ ≤ Big O

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots < 2^n$

lower bound | Avg. bound | upper bound



RULES: 1) keep track of # of permutations for each node, each left node = 1 possible perm., 2) each node w/ one child = # of perms of that child 4) each node w/ 2 children = $L * R * P(L,R)$

L = # of perms of left child

R = # of perms of right child

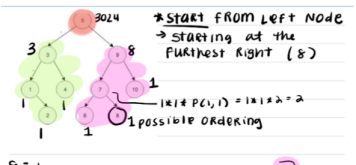
L = # of nodes in left subtree

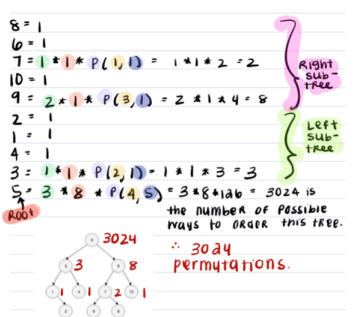
R = # of nodes in right subtree

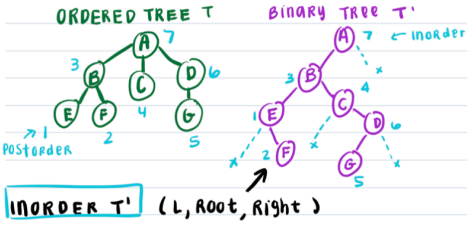
P = is a function i'll define

5) $P(n,m) = \frac{(n+m)!}{m!\,n!}$ ← # of ways to permutate all the nodes in the left & right subtrees

gets rid of all the permutations that don't have the nodes in a proper sequence that would work.







INORDER T' (L, ROOT, RIGHT)

E - F - B - C - G - D - A

1 2 3 4 5 6 7

PREORDER = (ROOT, LEFT, RIGHT)

A - B - E - F - C - D - G

INORDER = (LEFT, ROOT, RIGHT)

E - B - F - A - C - G - D

POSTORDER = (LEFT, RIGHT, ROOT)

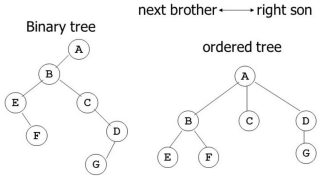
E - F - B - C - G - D - A

1 2 3 4 5 6 7

Ordered tree and binary tree

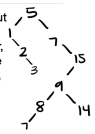
- An ordered tree can be represented as a binary tree.

oldest son ⟷ left son
next brother ⟷ right son



5 -50

To create a binary search tree from an array, go through it without sorting and then compare ***each*** element in the array to the root and if it is ***less*** than the root, it becomes a ***left*** child, if it is larger, then it become a ***right*** child, continuing comparing as you move down the tree for proper insertion.BTS do ***not*** have to balanced, and the O time for finding an element in the tree = *O(log n)* and inserting is also *O(log n)*. Binary trees have max of 2 children.
Example: array[5, 7, 1, 15, 9, 2, 14, 8, 7, 3]



An algorithm takes $0.4ms$ for input size 50 (this allows you to determine the constant c, which will be different in each case). How large of an input can be solved in one hour if the run time of the algorithm is ... ?

(a) First we determine the constant: $c \cdot 50 = 0.4$ or $c = 0.4/50$. Now plug it in and solve for 1 hour $(3,600,000ms)$:

$$c \cdot n = \frac{0.4}{50} \cdot n = 3,600,000$$

$$n = 3,600,000 \cdot \frac{50}{0.4} = 450,000,000$$

(b) Constant (note that logarithm base does not matter, but here we'll use base 2): $c \cdot 50 \lg 50 = 0.4$ or $c = \frac{0.4}{50 \lg 50}$.

$$c \cdot n \lg n = \frac{0.4}{50 \lg 50} 3600000$$

$$n \lg n = 3600000 \cdot \frac{50 \lg 50}{0.4} \approx 2,539,735,285.4$$

Now, since $n \lg n$ doesn't have a nice inverse, we have to try different values for n to narrow it down.

$$300,000,000 \lg(300,000,000) \approx 8,448,116,177.9 \text{ (too big)}$$

$$30,000,000 \lg(30,000,000) \approx 745,000,000 \text{ (too small)}$$

etc... to get n in the range

$$95,750,000 < n < 95,800,000$$

(c) Constant: $c \cdot 50^3 = 0.4$ or $c = \frac{0.4}{50^3}$.

$$c \cdot n^3 = \frac{0.4}{50^3} \cdot n^3 = 3,600,000$$

$$n^3 = 3600000 \cdot \frac{50^3}{0.4} = 1,125,000,000,000$$

$$n = \lfloor (1,125,000,000,00)^{1/3} \rfloor = 10400$$

(d) Constant: $c \cdot 50 = 0.4$ or $c = \frac{0.4}{2^{50}}$.

$$c \cdot 2^n = \frac{0.4}{2^{50}} \cdot 2^n = 3600000$$

$$2^n = 3600000 \cdot \frac{2^{50}}{0.4}$$

$$n = \lfloor \lg(3600000 \cdot \frac{2^{50}}{0.4}) \rfloor = \lfloor \lg(3600000) + \lg(2^{50}) - \lg(0.4) \rfloor$$

$$\ldots = \lfloor 21.779 + 50 + 1.322 \rfloor = 83$$

(a) Horner's rule takes $\Theta(n)$ time to evaluate a polynomial.
(b) The extra-naïve algorithm would take $\Theta(n^2)$ steps:

```
poly=0
for k=0 to n
    term = a_k
    for i=1 to k
        term *= x
    poly += term
return poly
```

(c) Let's call the loop invariant $\gamma(i, y)$, which is true when

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

initialization Upon entering the loop of the given Horner's rule implementation, $y = 0$ and $i = n$. Checking: $\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k = \sum_{k=0}^{n-(n+1)} a_{k+n+1} x^k = \sum_{k=0}^{-1} a_{k+n+1} x^k = 0 = y$, so $\gamma(n, 0)$ is trivially true.

maintenance Here as with induction, we look at the effect of one iteration of the loop. Suppose that $\gamma(i, y)$ is true for some $i > 0$. After one time through the loop, the new value of i will be $i' = i - 1$ and y will be updated to y' according the the formula in the loop. We need to show that $\gamma(i', y')$ is true - in other words, the loop maintains the invariant γ

The loop body sets $y' = a_i + x \cdot y$ and we know from $\gamma(i, y)$ that $y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$. Now

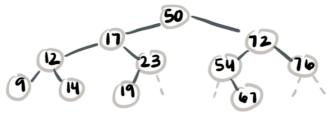
$$\begin{aligned} y' &= a_i + x \cdot y \\ &= a_i + x \cdot \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \\ &= a_i + x \cdot (a_{i+1} x^0 + a_{i+2} x^1 + a^{i+3} x^2 + \cdots + a_n x^{n-(i+1)}) \\ &= a_i + (a_{i+1} x^1 + a_{i+2} x^2 + a^{i+3} x^3 + \cdots + a_n x^{n-(i+1)+1}) \\ &= a_i x^0 + a_{i+1} x^1 + a_{i+2} x^2 + a^{i+3} x^3 + \cdots + a_n x^{n-i} \\ &= a_{i'+1} x^0 + a_{i'+2} x^1 + a^{i'+3} x^2 + a^{i'+4} x^3 + \cdots + a_n x^{n-(i'+1)} \\ &= \sum_{k=0}^{n-(i'+1)} a_{k+i'+1} x^k \end{aligned}$$

and therefore $\gamma(i', y')$ is true.

(d) At termination, $\gamma(i, y)$ is true and $i = -1$. Plugging that into the formula we see

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k = \sum_{k=0}^{n-(-1+1)} a_{k-1+1} x^k = \sum_{k=0}^{n} a_k x^k$$

which is what was intended.



SUCCESSOR = smallest value greater than the node.

1) if has right subtree, then it is the left most leaf. Ex: SUC(17) = 19
2) NO right subt., node is right child, go up the tree until you get to a Right chain, then it is the value the Right chain points to Ex: SUC(23) = 50
3) NO right subt., node is left child, it is the parent of that node. Ex: SUC(19) = 23

PREDECESSOR = largest value smaller than the node.

1) if has left subt., then it's the Right most leaf. Ex: PRED(17) = 14
2) NO left subt., node is left child, go up the tree until left pointing chain, the it is that value. Ex: PRED(19) = 17
3) NO left subt., node is Right child, it is the parent of that node. Ex: PRED(76) = 72

```
input: integer n>=0
output: integer k, array b of k bits

convert_to_binary(n)   {

-- initialization
int k=0
int t=n
array b = [] of bit

--loop
while t>0 do
      b[k] = (t mod 2)
      k = k+1
      t = t div 2

--end
return b, k
}
```

loop condition = γ = "$t > 0$"

loop invariant α used:

$t \geq 0$

$m = \sum_{i=0}^{k-1} b[i] \cdot 2^i$ be the # represented by b.

Then $n = 2^k \cdot t + m$

Initialization: α is true after initialization phase

Initially, $t = n \geq 0$, so part 1 of α is true.
Also, $k = 0$ and $b = []$, so $m = 0$
so part 2 is true since $2^k \cdot t + m = 2^0 \cdot n + 0 = n$

Maintenance: suppose γ is true so that the loop can be entered. If α is true, then after one execution of the body of the loop $\mathcal{L}$, the invariant α will still be true.

Suppose both γ and α are true. Then $t > 0$ and $n = 2^k \cdot t + m$ (where m is defined above). The new values of t, k, m we will call t', k', m' and clearly $k' = k + 1$. Also, $t' = \lfloor t/2 \rfloor$. Need to show that α holds for these new value t', k', m'. Part 1 of α is easy: $t > 0$ so $t' = \lfloor t/2 \rfloor \geq 0$.
For part 2, it remains to show that $n = 2^{k'} \cdot t' + m'$
There are two cases, depending on whether t is even or odd.
(t is even) Here $b[k] = 0$, $m' = m$, and $t' = t/2$ Now,

$$2^{k'} \cdot t' + m' = 2^{k+1} \cdot \frac{t}{2} + m = 2^k \cdot t + m = n$$

(t is odd) $b[k] = 1$, $m' = 2^k + m$, and $t' = \frac{t-1}{2}$ so,

$$2^{k'} \cdot t' + m' = 2^{k+1} \cdot \frac{t-1}{2} + (2^k + m) = 2^k \cdot t - 2^k + (2^k + m) = 2^k \cdot t + m = n$$

and so part 2 of α is true.

termination: Eventually γ will occur, so the loop will halt. Desired outcome: $\gamma \wedge \alpha$

The loop terminates since $t > \lfloor t/2 \rfloor$
At termination, both α and γ are true. Part 1 of α tells us that $t \geq 0$ while γ says that $t \leq 0$. From these we get $t = 0$

Now lets look at part 2 of α at termination. Remember that $m = \sum_{i=0}^{k-1} b[i] \cdot 2^i$ is the # represented by bits stored in the array b. Part 2 says that $n = 2^k \cdot t + m$. But we know that $t = 0$, so $n = m$. That is what we wanted to prove, namely that b stores the binary representation of the # n.

example

```
input: integer n>0
output: n(n+1)/2

--initialization
int s=0
int k=0

--loop
while k < n+1 do
    s = s+k
    k = k+1

--end
return s
```

γ: k < n+1

α:
- $0 \leq k \leq n+1$
- s = k(k-1)/2

invariant: α

initialization: show that α is true after the <init> phase of the code has been executed

maintenance: show that if $\alpha \wedge \gamma$ is true, then α will be true after one execution of the loop body $\mathcal{L}$

termination: the loop finishes when γ is false, so argue that $\neg\gamma \wedge \alpha$ is the desired outcome

code:

<init>
while γ
  do $\mathcal{L}$