We will have a quiz on Thursday over trees.

A. Abstract Syntax Trees

Cast an arithmetic expression as a tree with the last operator to be done as the root. The numbers or variables are leaf nodes.

Now if you print out in order you would see the operator in the right place, i.e. between the nodes, but what about precedence and parentheses?

What we should do is surround the output of a tree traversal with parentheses:

```python
def _inorder_aux(self, nodeptr):
    if nodeptr == None:
        return #nothing to visit!

    print ("(")
    self._inorder_aux(nodeptr.lchild)
    print(nodeptr.value)
    self._inorder_aux(nodeptr.rchild)
    print ")")
```

This puts too many parentheses in: 

\[(6 \times 2) + 3\]  and  \[((7 + 5) \times 4)\]

But to figure out whether you need parentheses would be pretty hard. What would you have to do?
A. Tree traversals (finish)

**Depth-first** (doesn't matter if it is pre-order, in-order or post-order)
Use a stack: push all children in reverse order

Depth-first is the most common way of looking at things.

**Breadth-first**
Use a queue: put all children into a queue

Breadth-first will often be used when you don't want to trace down to the very remotest leaf, which is the case in a game tree. First of all, it would take an unbelievably long time to go that far, like many lifetimes of the Universe! There are $10^{120}$ positions.

Depth-first uses a stack, and recursion implicitly uses a stack. Depth-first would use a queue:

- Make a queue for the node pointers
- Put the root in the queue
- while the queue is not empty:
  - Take the head item of the queue (dequeue)
  - Print out its value
  - If it has children nodes, put these children nodes on the queue (enqueue), first leftchild, then rightchild

- head
  - dog
  - ant
  - asp
  - zebra
  - ape
- cat

- search dir
directory files of file

folder

Multics operating system

GE-645

UNICS

X

DAG

find - depth-first tree walk

C:

H:

L:

forest
B. Heaps

A priority queue is

Why we would like a priority queue. But how to dequeue the highest priority item without searching the entire list? Use a heap!

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>O(log N)</td>
<td>O(log N)</td>
</tr>
<tr>
<td>BST</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

We could liken a heap to a weakly ordered tree, or a partially ordered tree.

Two kinds: minheap and maxheap (book shows minheap)

Heap property: the parent is greater than either of its children, but no relationship exists between leftchild and rightchild. That is, no ordering between them.

Requires that we have a complete binary tree.

Demo of heapsort: code

Exercise: do some percUp and percDown operations.

Go over the definition of heap, priority queue, and Binary Search Tree.

Talked about the

- complete tree property
- heap property
- binary search tree property

How are BSTs different from heaps?

Uses of heaps:

- heap sort
- priority queue – to get the next one out of the queue that needs to be (highest priority)

Time complexity of heapsort:

\[ T(n) = 2n \log_2(n) \]

The \( O(T(n)) = n \log n \)

Minheap – used for sorting into ascending order
Maxheap – used for sorting into descending order
1. A tree is a linked node structure. Define each of the following (2 pts).

<table>
<thead>
<tr>
<th>root</th>
<th>The root is the unique node that is at the &quot;top&quot; of the tree. It has no parent, no arrows coming into it.</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaf</td>
<td>A leaf is any node (there may be many) that has no children, i.e. no arrows coming out of it. It is possible for the root to be a leaf, if the tree has only one node.</td>
</tr>
</tbody>
</table>

2. Write the TreeNode class that would carry the payload of a node and had children. This is a binary tree. (2 pts)

```python
class TreeNode:
    def __init__(self):
        self.value = None
        self.left_child = None
        self.right_child = None
```

3. For binary search trees what does the sorted property require? (2 pts)

Given node N of the BST,

The value in N is greater than the value of any of the nodes in the left subtree of N.

The value in N is less than the value of any of the nodes in the right subtree of N.

4. For binary search trees what does the heap property require? (2 pts)

If this is a maximal heap, then the root has the largest value, and

Given node N of the BST,

The value in N is greater than the value of any of the nodes in the left subtree of N.

The value in N is greater than the value of any of the nodes in the right subtree of N.

If this is a minimal heap, then replace greater than with less than.

5. Draw a BST with four children. The root is elephant. Then insert in this order: ape, zebra, camel. (2 pts)

```
          elephant
          /     \
        ape    zebra
          \     /\    ...
        camel  2 4 1 2
```

- ape
- zebra
- camel
6.6 Priority Queues with Binary Heaps

In Chapter 3 you learned about the first in first out data structure called a queue. One important variation of a queue is called a priority queue. A priority queue acts like a queue in that you dequeue an item by removing it from the front. However, in a priority queue the logical order of items inside a queue is determined by their priority. The highest priority items are at the front of the queue and the lowest priority items are at the back. Thus when you enqueue an item on a priority queue, the new item may move all the way to the front. We will see that the priority queue is a useful data structure for some of the graph algorithms we will study in the next chapter.

You can probably think of a couple of easy ways to implement a priority queue using sorting functions and lists. However, inserting into a list is \( O(n) \) and sorting a list is \( O(n \log n) \). We can do better. The classic way to implement a priority queue is using a data structure called a binary heap. A binary heap will allow us both enqueue and dequeue items in \( O(\log n) \).

The binary heap is interesting to study because when we diagram the heap it looks a lot like a tree, but when we implement it we use only a single list as an internal representation. The binary heap has two common variations: the min heap, in which the smallest key is always at the front, and the max heap, in which the largest key value is always at the front. In this section we will implement the min heap. We leave a max heap implementation as an exercise.

6.6.1 Binary Heap Operations

The basic operations we will implement for our binary heap are as follows:

- `BinaryHeap()` creates a new, empty, binary heap.
- `insert(k)` adds a new item to the heap.
- `findMin()` returns the item with the minimum key value, leaving item in the heap.
- `delMin()` returns the item with the minimum key value, removing the item from the heap.
- `isEmpty()` returns true if the heap is empty; false otherwise.
- `size()` returns the number of items in the heap.
- `buildHeap(list)` builds a new heap from a list of keys.

The following Python session demonstrates the use of some of the binary heap methods.
def percDown(k):
    while k < self.currentsize:
        leftpos = 2 * k
        rightpos = 2 * k + 1
        leftvalue = heap[leftpos]
        rightvalue = heap[rightpos]
        if leftvalue < rightvalue:
            swap k and leftpos
            temp = heap(k)
            heap(k) = heap[leftpos]
            heap[leftpos] = temp
        else:
            k = leftpos
        k = rightpos
>>> from python3.scripts import BinaryHeap
>>> bh = BinaryHeap()
>>> bh.insert(5)
>>> bh.insert(7)
>>> bh.insert(3)
>>> bh.insert(11)
>>> print(bh.delMin())
3
>>> print(bh.delMin())
5
>>> print(bh.delMin())
7
>>> print(bh.delMin())
11

### 6.6.2 Binary Heap Implementation

#### 6.6.2.1 The Structure Property

In order to make our heap work efficiently, we will take advantage of the logarithmic nature of the tree to represent our heap. You will learn in section 6.7.3 that in order to guarantee logarithmic performance, we must keep our tree balanced. A balanced binary tree has roughly the same number of nodes in the left and right subtrees of the root. In our heap implementation we keep the tree balanced by creating a complete binary tree. A complete binary tree is a tree in which each level has all of its nodes. The exception to this is the bottom level of the tree, which we fill in from left to right. Figure 6.14 shows an example of a complete binary tree.

Another interesting property of a complete tree is that we can represent it using a single list. We do not need to use nodes and references or even lists of lists. Because the tree is complete, the left child of a parent (at position \( p \)) is the node that is found in position \( 2p \) in the list. Similarly, the right child of the parent is at position \( 2p + 1 \) in the list. To find the parent of any node in the tree, we can simply use Python’s integer division. Given that a node is at position \( n \) in the list, the parent is at position \( n / 2 \). Figure 6.15 illustrates a complete binary tree and also gives the list representation of the tree. The list representation of the tree, along with the full structure property, allows us to efficiently traverse a complete binary tree using only a few simple mathematical operations. We will see that this also leads to an efficient implementation of our binary heap.
6.6.2.2 The Heap Order Property

The method that we will use to store items in a heap relies on maintaining the heap order property. The heap order property is as follows: In a heap, for every node $x$ with parent $p$, the key in $p$ is smaller than or equal to the key in $x$. Figure 6.15 also illustrates a complete binary tree that has the heap order property.

6.6.2.3 Heap Operations

We will begin our implementation of a binary heap with the constructor. Since the entire binary heap can be represented by a single list, all the constructor will do is initialize the list and an attribute `currentSize` to keep track of the current size of the heap. Listing 6.17 shows the Python code for the constructor. You will notice that an empty binary heap has a single zero as the first element of `heapList` and that this zero is not used, but is there so that simple integer division can be used in later methods.

```python
def __init__(self):
    self.heapList = [0]
    self.currentSize = 0

Listing 6.17: Create a New Binary Heap
```
The next method we will implement is `insert`. The easiest, and most efficient, way to add an item to a list is to simply append the item to the end of the list. The good news about appending is that it guarantees that we will maintain the complete tree property. The bad news about appending is that we will very likely violate the heap structure property. However, it is possible to write a method that will allow us to regain the heap structure property by comparing the newly added item with its parent. If the newly added item is less than its parent, then we can swap the item with its parent. Figure 6.16 shows the series of swaps needed to percolate the newly added item up to its proper position in the tree.

Notice that when we percolate an item up, we are restoring the heap property between the newly added item and the parent. We are also preserving the heap property for any siblings. Of course, if the newly added item is very small, we may still need to swap it up another level. In fact, we may need to keep swapping until we get to the top of the tree. Listing 6.18 shows the `percUp` method, which percolates a new item as far up in the tree as it needs to go to maintain the heap property. Here is where our wasted element in `heapList` is important. Notice that we can compute the parent of any node by using simple integer division. The parent of the current node can be computed by dividing the index of the current node by 2.

When we reach 0 we know there is no parent to this node.
Figure 6.16: Percolate the New Node up to Its Proper Position
We are now ready to write the `insert` method. The Python code for `insert` is shown in Listing 6.19. Most of the work in the `insert` method is really done by `percUp`. Once a new item is appended to the tree, `percUp` takes over and positions the new item properly.

```python
def percUp(self, i):
    while i // 2 > 0:
        if self.heapList[i] < self.heapList[i // 2]:
            tmp = self.heapList[i // 2]
            self.heapList[i // 2] = self.heapList[i]
            self.heapList[i] = tmp
            i = i // 2
```

Listing 6.18: The `percUp` Method

```python
def insert(self, k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self.percUp(self.currentSize)
```

Listing 6.19: Adding a New Item to the Binary Heap

With the `insert` method properly defined, we can now look at the `delMin` method. Since the heap property requires that the root of the tree be the smallest item in the tree, finding the minimum item is easy. The hard part of `delMin` is restoring full compliance with the heap structure and heap order properties after the root has been removed. We can restore our heap in two steps. First, we will restore the root item by taking the last item in the list and moving it to the root position. Moving the last item maintains our heap structure property. However, we have probably destroyed the heap order property of our binary heap. Second, we will restore the heap order property by pushing the new root node down the tree to its proper position. Figure 6.17 shows the series of swaps needed to move the new root node to its proper position in the heap.

In order to maintain the heap order property, all we need to do is swap the root with its smallest child less than the root. After the initial swap, we may repeat the swapping process with a node and its children until the node is swapped into a position on the tree where it is already less than both children. The code for percolating a node down the tree is found in the `percDown` and `minChild` methods in Listing 6.20.
if we propagate a hole down, we might not end up with a complete BT

perDown

Figure 6.17: Percolating the Root Node down the Tree
def percDown(self, i):
    while (i * 2) <= self.currentSize:
        mc = self.minChild(i)
        if self.heapList[i] > self.heapList[mc]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
            i = mc

def minChild(self, i):
    if i * 2 + 1 > self.currentSize:
        return i * 2
    else:
        if self.heapList[i*2] < self.heapList[i*2+1]:
            return i * 2
        else:
            return i * 2 + 1

Listing 6.20: The percDown Method

The code for the delMin operation is in Listing 6.21. Note that once
again the hard work is handled by a helper function, in this case percDown.

def delMin(self):
    retval = self.heapList[1]
    self.currentSize = self.currentSize - 1
    self.heapList.pop() — removes last item
    self.percDown(1)
    return retval

Listing 6.21: Deleting the Minimum Item from the Binary Heap

To finish our discussion of binary heaps, we will look at a method to
build an entire heap from a list of keys. The first method you might think
of may be like the following. Given a list of keys, you could easily build
a heap by inserting each key one at a time. Since you are starting with a
list of one item, the list is sorted and you could use binary search to find
the right position to insert the next key at a cost of approximately $O(\log n)$
operations. However, remember that inserting an item in the middle of the
list may require $O(n)$ operations to shift the rest of the list over to make
room for the new key. Therefore, to insert \( n \) keys into the heap would require a total of \( O(n \log n) \) operations. However, if we start with an entire list then we can build the whole heap in \( O(n) \) operations. Listing 6.22 shows the code to build the entire heap.

```python
def buildHeap(self, alist):
    i = len(alist) // 2
    self.currentSize = len(alist)
    self.heapList = [0] + alist[:]
    while (i > 0):
        self.percDown(i)
    i = i - 1
```

**Listing 6.22: Building a New Heap from a List of Items**

![Diagram of heap construction](image)

**Figure 6.18: Building a Heap from the List \([9, 6, 5, 2, 3]\)**

Figure 6.18 shows the swaps that the `buildHeap` method makes as it moves the nodes in an initial tree of \([9, 6, 5, 2, 3]\) into their proper positions. Although we start out in the middle of the tree and work our way back toward the root, the `percDown` method ensures that the largest child is always moved down the tree. Because it is a complete binary tree, any nodes past the halfway point will be leaves and therefore have no children. Notice that when \( i = 1 \), we are percolating down from the root of the tree, so this may require multiple swaps. As you can see in the rightmost two subtrees of Figure 6.18, first the 9 is moved out of the root position, but after 9 is moved down one level in the tree, `percDown` ensures that we check the next set of children farther down in the tree to ensure that it is pushed as low as it can go. In this case it results in a second swap with 3. Now
that 9 has been moved to the lowest level of the tree, no further swapping can be done. It is useful to compare the list representation of this series of swaps as shown in Figure 6.19 with the tree representation.

\[
\begin{align*}
  i = 2 & \quad [0, 9, 5, 6, 2, 3] \\
  i = 1 & \quad [0, 9, 2, 6, 5, 3] \\
  i = 0 & \quad [0, 2, 3, 6, 5, 9]
\end{align*}
\]

**Figure 6.19:** Building a Heap from the List [9, 5, 6, 2, 3]

The assertion that we can build the heap in \(O(n)\) may seem a bit mysterious at first, and a proof is beyond the scope of this book. However, the key to understanding that you can build the heap in \(O(n)\) is to remember that the \(\log n\) factor is derived from the height of the tree. For most of the work in buildHeap, the tree is shorter than \(\log n\).

Using the fact that you can build a heap from a list in \(O(n)\) time, you will construct a sorting algorithm that uses a heap and sorts a list in \(O(n \log n)\) as an exercise at the end of this chapter.
To do heap sort

unsort numbers in a list
num list = \{27, 16, 5, 39, 99, 6\}

1. for each x in num list
   myheap.insert(x)

2. now you have a heap
   while heap is not empty
      remove root & print
      move last one to root
      percdown(1)

O(n log n) = n x 2 x log n
Informal Homework No. 2
Heaps

Complete these exercises in preparation for the exam.


```
\begin{tikzpicture}
  \node (root) {cat}
    child {node (A) {terret}
      child {node (A1) {giraffe}
        child {node {}}
        child {node {}}
      }
      child {node (A2) {fox}
        child {node {}}
        child {node {}}
      }
    }
    child {node (B) {dog}
      child {node (B1) {}
        child {node {}}
        child {node {}}
      }
      child {node (B2) {}
        child {node {}}
        child {node {}}
      }
    }
\end{tikzpicture}
```

Now put all these same letters into the array below.

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>terret</td>
<td>dog</td>
<td>giraffe</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

In the picture below, circle the node in level 1 (which is the topmost level). Notice that level 2's nodes are already circled. Then indicate level 3 and level 4 in the same way.

```
\begin{tikzpicture}
  \node (root) {2}
    child {node {}}
    child {node {}}
    child {node {}}
    child {node {}}
\end{tikzpicture}
```

In the array below, the top level is indicated by 1 in the lower box. The second level (right below the top level is labeled 2.

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```