Quiz 12 will be on Thursday will be about heaps, including percUp and percDown. Study the ACE from Day 26. Answers are in scan for day 26.

**A. Go over quiz 11**

2 different places to put the check for the base case. Python-like language:

```python
def visit_inorder(nodeptr)
    if nodeptr.lchild is not empty:
        visit_inorder(nodeptr.lchild)

    print (nodeptr.value)

    if nodeptr.rchild is not empty:
        visit_inorder(nodeptr.rchild)

end
```

```python
def visit_inorder(nodeptr)
    if nodeptr is empty then return

    visit_inorder(nodeptr.lchild)

    print (nodeptr.value)

    visit_inorder(nodeptr.rchild)

end
```

The following is more like the Scala language because it uses cases:

```scala
def visit_inorder(nodeptr)
    case nodeptr:
        empty => return
        not empty => visit_inorder(nodeptr.lchild)
                       print(nodeptr.value)
                       visit_inorder(nodeptr.rchild)

end
```
\[
\sum_{i=0}^{m} 2^i = 2^{m+1} - 1 \\
\]

**Inductive Proof**

\[2^0 = 1\] base case

\[\sum_{i=0}^{k} 2^i = 2^{k+1} - 1\] assume hypothesis

\[\sum_{i=0}^{k+1} 2^i = 2^{k+2} - 1\] inductive step 

\[\sum_{i=0}^{k+1} 2^i = \sum_{i=0}^{k} 2^i + 2^{k+1}\]

\[2^0 + 2^1 + 2^2 + \ldots + 2^k + 2^{k+1} = 2^{k+2} - 1\]

\[2^{k+1} - 1 = 2^{k+1} - 1 + 2^{k+1}\]

\[\sum_{i=0}^{k} 2^i = 2 \cdot 2^{k+1} - 1\]

\[= 2^{k+2} - 1\]
\begin{align*}
m &= 0 & 1 \\
m &= 1 & 3 = 1 + 2 \\
m &= 2 & 7 = 1 + 2 + 4 \\
m &= 3 & 15 = 1 + 2 + 4 + 8 \\
\end{align*}

\begin{align*}
\begin{array}{c|cccccccc}
  m & 0 & 1 & 2 & 3 & 4 & 5 & \ldots \\
\hline
  0 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
  1 & 3 & 1 & 2 & 1 & 2 & 1 & \ldots \\
  2 & 7 & 1 & 2 & 4 & 1 & 2 & 1 & \ldots \\
  3 & 15 & 1 & 2 & 4 & 8 & 1 & 2 & 1 & \ldots \\
  4 & 31 & 1 & 2 & 4 & 8 & 16 & 1 & 2 & 1 & \ldots \\
  5 & 63 & 1 & 2 & 4 & 8 & 16 & 32 & 1 & 2 & 1 & \ldots \\
\end{array}
\end{align*}
4. You are given a tree stored in memory using the standard method. There is a node at position \( p \) where \( p = 1 \) on up. Where are \( p \)'s 2 children? (1 pt)

\[ 2p, 2p+1 \]

5. Given a complete binary tree that has \( N \) nodes in it, total, what is the length of the longest path as a function of \( N \)? (1 pt)

\[ 2^x = y \]

\[ \log_2 y = x \]

\[ [\log_2 N] \]

6. You are given a complete binary tree that has \( m \) levels. (A tree with only a root is considered to have 0 levels. If there are 3 children, it has one level, and so forth.)

a.) How many nodes are there at level \( k \)? The root is at level \( k \). (1 pt)

\[ 2^k \] nodes

b.) How many total nodes are there as a function of \( m \), where \( m \) is the total number of levels? Write a closed form formula. (1 pt)

\[ \sum_{k=0}^{m} 2^k = 2^{m+1} - 1 \]

Following are some lines that could be used to implement any of three different depth-first tree traversals.

```python
def _traverse_Xorder_aux(self, nodeptr):
    1 print(nodeptr.value + "",")
    2 self._traverse_Xorder_aux(nodeptr.rchild)
    3 self._traverse_Xorder_aux(nodeptr.lchild)
    4 if nodeptr == None: return
```

7. Show what the order would be for post order traversal. Arrange the line numbers 1-4 below. (1 pt)

\[ 4 \quad 3 \quad 2 \quad 1 \]

8. Show what the order would be for in order traversal. Arrange the line numbers 1-4 below. (1 pt)

(the order of the traversal should be natural for a left to right scan. That is, the left child is visited before the right.)

\[ 4 \quad 3 \quad 1 \quad 2 \]
B. Dictionaries

Also called maps or associative arrays.

Can be implemented in many ways. You can implement it crudely using an array of tuples but the search times may be horribly slow, on the order of N. Brute force implementation, or "reference" implementation. Of course you could store them sorted by key and then use binary search on the keys to speed up search.

Just imagine a map of 330 million SSNs, which is a key for people. Searching linearly each time would be horrible. Or license plates at border patrol (e.g. in Fort Erie). Or fingerprints! Each fingerprint could be turned into a very long binary number (just scan the bitmap image and each line is part of the number.)

But generally you want to balance both update and search operations so that neither is terribly slow.

http://paulbourke.net/dataformats/bitmaps/

So we can implement using various space-saving and time-saving techniques. Project 4 is one example. Best use of space, but slow because traversing a linked list may be very slow! This is why we use the speedups array to cache recent references.

However, this benefits us only if we see locality exhibited in the data access patterns. That is, you tend to look at only a few specific spots. If you search randomly all over the data space, then caching is worthless.
C. Hashing

Hashing the way of taking in some value and transforming it to another value:

Desired properties:
- No 2 input values hash to the same output value (no collisions)
- The output values are evenly distributed and often far apart
- It is impossible to go from the output value to the input value (security)
- The output value is usually smaller than the input, sometimes much much smaller. (it has to be if it turns into an index number!)

This is a meat grinder but my grandpa used it to grind up potatos, carrots, onions and horseradish. The concoction that came out was a mushy mess called "hash" that he fried in a pan and tasted wonderful!

Hashing is used in several other areas:
- for security, to scramble a password. Unix stores the scrambled pw instead of the actual password. When a user tries to login, their input is then scrambled and compared to the stored scrambled password. If match, then they are authenticated. If not, they are not allowed in.
- for quick validity verification. If you have two very long documents, comparing them for equality might be tedious and time consuming. But if you know what their checksums are supposed to be, then you compare two large numbers (e.g. 40 digits long) and that's all. The checksum is a number derived from the text by some scheme (e.g. adding up all the ASCII codes.)

hashing algorithms:

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prime number modulus</td>
<td>non-prime number modulus</td>
</tr>
<tr>
<td>Complex algorithm (but not too time consuming!)</td>
<td>Takes too long to compute the hash</td>
</tr>
</tbody>
</table>
ACAT
A CAT
65 32 67 65 84

8 bit
01000001
00100000

horrible hash value
add up all ASCII values % 2

i 2 3 4
MARK
77 65 82 75

289
1 2 3 4
KRAM

77 + 65*2 + 82*3 + 75*4
75*1 + 82*2 + 65*3 + 77*4
D. Other implementations of dicts

The standard way is to use a large fixed size array and then a hash function to direct where the
data points should be stored. Each slot in the array is called a bucket. Conceptually it can be just
one location, but it could also be a small set, an area that is searched linearly. Typically, a
bucket is 1, 2, 4, or less than 10 items.

But what happens when 2 different data points hash to the same spot?

\[ H("Kathy") = 1 \]
\[ H("Sally") = 1 \]

Then we have a collision.

Collision = bucket overflow

If the bucket size = 1, then 2 different data points colliding = problems!

Approaches to collision resolution:

1. Rehashing – putting into a different spot using a different hashing mechanism
2. Linear probing – finding the next available spot in the array, can degenerate into O(N)
3. Using a general overflow area – can degenerate into O(N)
4. Using special overflow areas for each bucket – not much different than making the
   buckets > 1

Let's look at linear probing.

Backup your work!!!

Funny story about backups

https://nakedsecurity.sophos.com/2016/04/15/sysadmin-snafu-flushes-whole-company/
\[ \text{MAP} \]

A function \( f \)

\[ \text{domain} \rightarrow \text{range} \]

\[ 'c' \rightarrow 97 \]
\[ 'a' \rightarrow 26 \]
\[ 'b' \]

\[ \text{enumerate with a long table if finite domain} \]

\[ \text{1 to 1 and onto} \]

\[ \mathbb{R} \rightarrow \mathbb{R} \]

\[ \sqrt{-1} \]
buffer overflow error

```
char name[10];
char [] name = new char[10];
```

Vulnerability 2002

TRADE-OFF

```
C++
```

<table>
<thead>
<tr>
<th>No bounds check</th>
<th>C++</th>
<th>takes time</th>
<th>secure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Java, Pascal</td>
<td>slow</td>
<td>secure</td>
</tr>
</tbody>
</table>
Fixes

1. expand # buckets
2. get better $H()$
3. get better overflow
If the number of buckets is huge, then much space is wasted. Why not reuse those empty buckets to store?