1. The textbook solves the coin-change problem with the following code (note the “set-builder-like” notation):
\[\{c \mid c \in \text{coinValueList} \text{ and } c \leq \text{change}\}\]

```
def recMC(change, coinValueList):
global backtrackingNodes
backtrackingNodes += 1
minCoins = change
if change in coinValueList:
    return 1
else:
    for i in [c for c in coinValueList if c <= change]:
        numCoins = 1 + recMC(change - i, coinValueList)
        if numCoins < minCoins:
            minCoins = numCoins
    return minCoins
```

Results of running this code:

- Change Amount: 63
- Coin types: [1, 5, 10, 25]
- Run-time: 70.689 seconds
- Fewest number of coins: 6
- Number of Backtracking Nodes: 67,716,925

I removed the fancy set-builder notation and replaced it with a simple `if`-statement check:

```
def recMC(change, coinValueList):
    global backtrackingNodes
    backtrackingNodes += 1
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in coinValueList:
            if i <= change:
                numCoins = 1 + recMC(change - i, coinValueList)
                if numCoins < minCoins:
                    minCoins = numCoins
            return minCoins
```

Results of running this code:

- Change Amount: 63
- Coin types: [1, 5, 10, 25]
- Run-time: 45.815 seconds
- Fewest number of coins: 6
- Number of Backtracking Nodes: 67,716,925

a) Why is the second version so much “faster”?

b) Why does it still take a long time?

2. To speed the recursive backtracking algorithm, we can prune unpromising branches. The general recursive backtracking algorithm for optimization problems (e.g., fewest number of coins) looks something like:

Backtrack(recursionTreeNode p) {
    for each child c of p do
        if promising(c) then
            if c is a solution that’s better than best then
                best = c
            else
                Backtrack(c)
        end if
    end for
}

General Notes about Backtracking:
- The depth-first nature of backtracking only stores information about the current branch being explored on the run-time stack, so the memory usage is “low” even though the # of recursion tree nodes might be exponential (2^n).
- Each node of the search-space (recursive-call) tree maintains the state of a partial solution. In general, the partial solution state consists of potentially large arrays that change little between parent and child. To avoid having multiple copies of these arrays, a reference to a single “global” array can be maintained which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.

a) For the coin-change problem, what defines the current state of a search-space tree node?
b) When would a “child” tree node NOT be promising?

3. Consider the output of running the backtracking code with pruning (next page) twice with a change amount of 63 cents.

<table>
<thead>
<tr>
<th>Change Amount: 63 Coin types: [1, 5, 10, 25]</th>
<th>Change Amount: 63 Coin types: [25, 10, 5, 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time: 0.036 seconds</td>
<td>Run-time: 0.003 seconds</td>
</tr>
<tr>
<td>Fewest number of coins 6</td>
<td>Fewest number of coins 6</td>
</tr>
<tr>
<td>The number of each type of coins is:</td>
<td>The number of each type of coins is:</td>
</tr>
<tr>
<td>number of 1-cent coins is 3</td>
<td>number of 25-cent coins is 2</td>
</tr>
<tr>
<td>number of 5-cent coins is 0</td>
<td>number of 10-cent coins is 1</td>
</tr>
<tr>
<td>number of 10-cent coins is 1</td>
<td>number of 5-cent coins is 0</td>
</tr>
<tr>
<td>number of 25-cent coins is 2</td>
<td>number of 1-cent coins is 3</td>
</tr>
<tr>
<td>Number of Backtracking Nodes: 4831</td>
<td>Number of Backtracking Nodes: 310</td>
</tr>
</tbody>
</table>

a) Explain why ordering the coins from largest to smallest produced faster results.

b) For coins of [50, 25, 12, 10, 5, 1] typical timings:

<table>
<thead>
<tr>
<th>Change Amount</th>
<th>Run-Time (seconds)</th>
<th>Number of Tree Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>399</td>
<td>8.88</td>
<td>2,015,539</td>
</tr>
<tr>
<td>409</td>
<td>55.17</td>
<td>12,093,221</td>
</tr>
<tr>
<td>419</td>
<td>318.56</td>
<td>72,558,646</td>
</tr>
</tbody>
</table>

Why the exponential growth in run-time?

4. As with Fibonacci, the coin-change problem can benefit from dynamic program since it was slow due to solving the same problems over-and-over again. Recall the general idea of dynamic programming:
   • Solve smaller problems before larger ones
   • store their answers
   • look-up answers to smaller problems when solving larger subproblems, so each problem is solved only once

a) To solve the coin-change problem using dynamic programming, we need to answer the questions:
   • What is the smallest problem?
   • Where do we store the answers to the smaller problems?
Data Structures

def backtrack(changeAmt, numberOfEachCoinType, numberOfCoinsSoFar, solutionFound, bestFewestCoins, bestNumberOfEachCoinType):
    global backtrackingNodes
    backtrackingNodes += 1
    for index in range(len(coinTypes)):
        smallerChangeAmt = changeAmt - coinTypes[index]
        if promising(smallerChangeAmt, numberOfCoinsSoFar + 1, solutionFound, bestFewestCoins):
            if smallerChangeAmt == 0:  # a solution is found
                if (not solutionFound) or numberOfCoinsSoFar + 1 < bestFewestCoins:  # check if its best
                    bestFewestCoins = numberOfCoinsSoFar + 1
                    bestNumberOfEachCoinType = [1] + numberOfEachCoinType
                    bestNumberOfEachCoinType[index] += 1
                    solutionFound = True
            else:
                # call child with updated state information
                smallerChangeAmtNumberOfEachCoinType = [1] + numberOfEachCoinType
                smallerChangeAmtNumberOfEachCoinType[index] += 1

                solutionFound, bestFewestCoins, bestNumberOfEachCoinType = backtrack(smallerChangeAmt, smallerChangeAmtNumberOfEachCoinType, numberOfCoinsSoFar + 1, solutionFound, bestFewestCoins, bestNumberOfEachCoinType)

        return solutionFound, bestFewestCoins, bestNumberOfEachCoinType

    def promising(changeAmt, numberOfCoinsReturned, solutionFound, bestFewestCoins):
        if changeAmt < 0:
            return False
        elif changeAmt == 0:
            return True
        else:  # changeAmt > 0
            if solutionFound and numberOfCoinsReturned + 1 >= bestFewestCoins:
                return False
            else:
                return True

    # Body of solveCoinChange
    numberOfEachCoinType = []  # set-up initial “current state” information
    numberOfCoinsSoFar = 0
    solutionFound = False
    bestFewestCoins = -1
    bestNumberOfEachCoinType = None

    numberOfEachCoinType = []
    for coin in coinTypes:
        numberOfEachCoinType.append(0)
    numberOfCoinsSoFar = 0
    solutionFound = False
    bestFewestCoins = -1
    bestNumberOfEachCoinType = None

    solutionFound, bestFewestCoins, bestNumberOfEachCoinType = backtrack(changeAmt, numberOfEachCoinType, numberOfCoinsSoFar, solutionFound, bestFewestCoins, bestNumberOfEachCoinType)

    return bestFewestCoins, bestNumberOfEachCoinType
Dynamic Programming Coin-change Algorithm:

I. Fills an array fewestCoins from 0 to the amount of change. An element of fewestCoins stores the fewest number of coins necessary for the amount of change corresponding to its index value.

For 29-cents using the set of coin types \{1, 5, 10, 12, 25, 50\}, the dynamic programming algorithm would have previously calculated the fewestCoins for the change amounts of 0, 1, 2, ..., up to 28 cents.

II. If we record the best, first coin to return for each change amount (found in the “minimum” calculation) in an array bestFirstCoin, then we can easily recover the actual coin types to return.

\[
fewestCoins[29] = \text{minimum}(fewestCoins[28], fewestCoins[24], fewestCoins[19], fewestCoins[17], fewestCoins[4]) + 1 = 2 + 1 = 3
\]

![Diagram of fewestCoins and bestFirstCoin arrays]

Extract the coins in the solution for 29-cents from bestFirstCoin[29], bestFirstCoin[24], and bestFirstCoin[12]

b) Extend the lists through 32-cents.

![Extended lists for fewestCoins and bestFirstCoin]

c) What coins are in the solution for 32-cents?
Objective: To gain experience with a simple, recursive, divide-and-conquer implementation and how to develop faster solutions using an iterative dynamic programming implementation.

To start the lab: Download and unzip the file at: www.cs.uni.edu/~fienup/cs1520f16/labs/lab6.zip

Part A: We've seen several recursive "divide-and-conquer" algorithms: fibonacci and coin-change problem. The general idea of divide-and-conquer algorithms is:
- dividing the original problem into small problem(s) (e.g., fib(n-1) and fib(n-2))
- solving the smaller problem(s) recursively
- combine the solution(s) to smaller problem(s) to solve the original problem (e.g., return fib(n-1) + fib(n-2))

Mathematics has several simple recursively defined functions. For example, the factorial function can be recursively defined as:

\[ n! = n \times (n - 1)! \quad \text{for } n \geq 1, \text{ and } \]
\[ 0! = 1 \]

Implement a recursive factorial(n) function using this recursive definition and test it with several small examples (e.g., 3! = 3*2*1 = 6, 4! = 4*3*2*1 = 24, and 5! = 5*4*3*2*1 = 120).

In Discrete Structures (CS 1800) you used (or will use) the binomial coefficient formula:

\[ C(n, k) = \frac{n!}{k!(n-k)!} \]

to calculate the number of combinations of "n choose k," i.e., the number of ways to choose k objects from n objects. For example, when calculating the number of unique 5-card hands from a standard 52-card deck (e.g., C(52, 5)) we need to calculate 52! / 5! 47! = 2,598,960.

Implement the function C(n, k) using formula (2) above and your recursive factorial function.

Completed your factorial and binomial coefficient functions, raise your hand and demonstrate your code.

Part B: One problem with using formula (2) in most languages is that n! grows very fast and overflows the integer representation before you can do the division to bring the value back to a value that can be represented.

(NOTE: Python does not suffer from this problem, but lets pretend that it does.)

For example, when calculating C(52, 5) we need to calculate 52! / 5! 47!. However, the value of

\[ 52! = 80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,883,277,824,000,000,000,000 \]

is much, much bigger than can fit into a 64-bit integer representation. Fortunately, another way to view C(52, 5)

is recursively by splitting the problem into two smaller problems by focusing on:
- the hands containing a specific card, say the ace of clubs, and
- the hands that do not contain the ace of clubs.

For those hands that do contain the ace of clubs, we need to choose 4 more cards from the remaining 51 cards, i.e., C(51, 4). For those hands that do not contain the ace of clubs, we need to choose 5 cards from the remaining 51 cards, i.e., C(51, 5). Therefore, C(52, 5) = C(51, 4) + C(51, 5).

In general,

\[ C(n, k) = C(n - 1, k - 1) + C(n - 1, k) \quad \text{for } 1 \leq k \leq (n - 1), \text{ and } \]
\[ C(n, k) = 1 \quad \text{for } k = 0 \text{ or } k = n \]

Implement the recursive "divide-and-conquer" binomial coefficient function using equation (3). Call your function DC(n, k) for "divide-and-conquer". Notice the difference in run-time between calculating the binomial coefficient using C(24, 12) vs. DC(24, 12), C(26, 13) vs. DC(26, 13), and C(28, 14) vs. DC(28, 14).

Completed your binomial coefficient function, DC, raise your hand and demonstrate your code.

Part C: Much of the slowness of your "divide-and-conquer" binomial coefficient function, DC(n, k), is due to redundant calculations performed due to the recursive calls. For example, the recursive calls associated with DC(5, 3) = 10 would be:
Pascal’s triangle (named for the 17th-century French mathematician Blaise Pascal, and for whom the programming language Pascal was also named) is a “dynamic programming” approach to calculating binomial coefficients.

\[
\begin{array}{ccccccc}
& & & & & 1 & \\
& & & & 1 & 1 & \\
& & & 1 & 2 & 1 & \\
& & 1 & 3 & 3 & 1 & \\
& 1 & 5 & 10 & 10 & 5 & 1 \\
\end{array}
\]

Recall that dynamic programming solutions eliminate the redundancy of recursive divide-and-conquer algorithms by calculating the solutions to smaller problems first, storing their answers, and looking up their answers if later needed instead of recalculating it. Abstractly, Pascal’s triangle relates to the binomial coefficient as in:

\[
\begin{array}{cccc}
C(0,0) & C(1,1) & C(2,2) & \cdots \\
C(1,0) & C(2,1) & C(3,3) & \cdots \\
C(2,0) & C(3,2) & C(4,4) & \cdots \\
\vdots & \vdots & \vdots & \ddots \\
C(n-1,k-1) & C(n-1,k) & C(n,k) & \cdots & C(n,n) \\
\end{array}
\]

For Part C, your job is to implement the “dynamic programming” binomial coefficient function using a Python list of lists to store Pascal’s triangle and loops (no recursion needed). Call your function \texttt{DP(n, k)} for “dynamic programming”. **Hints for Part C:**

* Review the dynamic programming fibonacci example from Lecture 9. File \texttt{lab6/fibonacci.py} contains the recursive divide-and-conquer, and two dynamic programming versions of fibonacci. The first dynamic programming version, \texttt{fib_DP}, stores the answers to all of the smaller problems. The second dynamic programming version, \texttt{fib_DP2}, reduces the amount of memory used by only storing the answers for the previous two smaller problems.

Notice the difference in run-time between calculating the binomial coefficient using \texttt{DC(24, 12)} vs. \texttt{DP(24, 12)}, \texttt{DC(26, 13)} vs. \texttt{DP(26, 13)}, and \texttt{DC(28, 14)} vs. \texttt{DP(28, 14)}.

**Completed your binomial coefficient function, DP, raise your hand and demonstrate your code.**

**Part D: EXTRA CREDIT**

* Your function \texttt{DP_Extra_Credit(n, k)} should use the idea of \texttt{fib_DP2} to avoid storing all the answers to the smaller problems. **Notice that the calculation of the next row in the picture above only needs the previous row and none of the older rows.**
1. Consider the following sequential search (linear search) code:

<table>
<thead>
<tr>
<th>Textbook’s Listing 5.1</th>
<th>Faster sequential search code</th>
</tr>
</thead>
<tbody>
<tr>
<td>def sequentialSearch(alist, item):</td>
<td>def linearSearch(alist, target):</td>
</tr>
<tr>
<td>&quot;&quot;&quot; Sequential search of unordered list &quot;&quot;&quot;</td>
<td>&quot;&quot;&quot; Returns the index of target in alist or -1 if target is not in alist&quot;&quot;&quot;</td>
</tr>
</tbody>
</table>
| pos = 0 | for position in range(len(alist)):
| found = False | if target == alist[position]:
| while pos < len(alist) and not found: | return position
| if alist[pos] == item: | return -1
| found = True | else:
| else: | if not breakOut:
| pos = pos+1 | return -1
| | else:
| | if target == alist[position]:
| | return position
| | else:
| | return -1
| | break
| | breakOut = True
| | stop = False
| | return found
| return found |
| |

(a) What is the basic operation of a search?

(b) For the following alist value, which target value causes linearSearch to loop the fewest ("best case") number of times?

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>aList:</td>
<td>10</td>
<td>15</td>
<td>28</td>
<td>42</td>
<td>60</td>
<td>69</td>
<td>75</td>
<td>88</td>
<td>90</td>
<td>93</td>
<td>97</td>
</tr>
</tbody>
</table>
```

(c) For the above alist value, which target value causes linearSearch to loop the most ("worst case") number of times?

d) For a successful search (i.e., target value in alist), what is the "average" number of loops?

<table>
<thead>
<tr>
<th>Textbook’s Listing 5.2</th>
<th>Faster sequential search code</th>
</tr>
</thead>
</table>
| def orderedSequentialSearch(alist, item): | def linearSearchOfSortedList(target, alist):
| """ Sequential search of ordered list """ | """ Returns the index position of target in sorted alist or -1 if target is not in alist""" |
| pos = 0 | breakOut = False |
| found = False | for position in range(len(alist)):
| stop = False | if target <= alist[position]:
| while pos < len(alist) and not found and not stop: | breakOut = True
| if alist[pos] == item: | break
| found = True | |
| else: | |
| if alist[pos] > item: | else:
| stop = True | if target == alist[position]:
| else: | return position
| pos = pos+1 | return -1
| | else:
| |
| | return -1
| | return found |
| return found |
| |

e) The above version of linear search assumes that alist is sorted in ascending order. When would this version perform better than the original linearSearch at the top of the page?
2. Consider the following binary search code:

<table>
<thead>
<tr>
<th>Textbook's Listing 5.3</th>
<th>Faster binary search code</th>
</tr>
</thead>
<tbody>
<tr>
<td>def binarySearch(alist, item):</td>
<td></td>
</tr>
<tr>
<td>first = 0</td>
<td></td>
</tr>
<tr>
<td>last = len(alist)-1</td>
<td></td>
</tr>
<tr>
<td>found = False</td>
<td></td>
</tr>
<tr>
<td>while first&lt;=last and not found:</td>
<td></td>
</tr>
<tr>
<td>midpoint = (first + last)//2</td>
<td></td>
</tr>
<tr>
<td>if alist[midpoint] == item:</td>
<td></td>
</tr>
<tr>
<td>found = True</td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>if item &lt; alist[midpoint]:</td>
<td></td>
</tr>
<tr>
<td>last = midpoint-1</td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>first = midpoint+1</td>
<td></td>
</tr>
<tr>
<td>return found</td>
<td></td>
</tr>
<tr>
<td>def binarySearch(target, lyst):</td>
<td></td>
</tr>
</tbody>
</table>
| """Returns the position of the target item if found, or -1 otherwise."""
| left = 0 |
| right = len(lyst) - 1 |
| while left <= right: |
| midpoint = (left + right) // 2 |
| if target == lyst[midpoint]: |
| return midpoint |
| elif target < lyst[midpoint]: |
| right = midpoint - 1 |
| else: |
| left = midpoint + 1 |
| return -1 |

a) "Trace" binary search to determine the worst-case basic total number of comparisons:

<table>
<thead>
<tr>
<th>loop #</th>
<th>worst-case elements remaining &quot;n&quot;</th>
<th>left</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>midpoint</th>
<th>right n-1</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>151</td>
<td></td>
</tr>
</tbody>
</table>

midpoint

b) What is the worst-case big-oh for binary search?

c) What is the best-case big-oh for binary search?

d) What is the average-case (expected) big-oh for binary search?

e) If the list size is 1,000,000, then what is the maximum number of comparisons of list items on a successful search?

f) If the list size is 1,000,000, then how many comparisons would you expect on an unsuccessful search?
3. Hashing Motivation and Terminology:
   a) Sequential search of an array or linked list follows the same search pattern for any given target value being searched for, i.e., scans the array from one end to the other, or until the target is found.
   If \( n \) is the number of items being searched, what is the average and worst case big-oh notation for a sequential search?
   - average case \( O(\quad) \)
   - worst case \( O(\quad) \)

   b) Similarly, binary search of a sorted array (or AVL tree) always uses a fixed search strategy for any given target value. For example, binary search always compares the target value with the middle element of the remaining portion of the array needing to be searched.
   If \( n \) is the number of items being searched, what is the average and worst case big-oh notation for a search?
   - average case \( O(\quad) \)
   - worst case \( O(\quad) \)

   **Hashing tries to achieve average constant time (i.e., \( O(1) \)) searching** by using the target’s value to calculate where in the array/Python list (called the hash table) it should be located, i.e., each target value gets its own search pattern. The translation of the target value to an array index (called the target’s home address) is the job of the hash function. A perfect hash function would take your set of target values and map each to a unique array index.

   **Set of Keys** | **Hash function** | **Hash Table Array**
   ------------- | ---------------- | -----------------------------------
   John Doe     | hash(John Doe) = 6 | 0  
   Philip East  | hash(Philip East) = 3 | 1  
   Mark Fienup  | hash(Mark Fienup) = 5 | 2  
   Ben Schafer  | hash(Ben Schafer) = 8 | 3  Philip East 3-2939 |
                      |                   | 4  Mark Fienup 3-5918 |
                      |                   | 5  John Doe 3-4567  |
                      |                   | 6  Ben Schafer 3-2187|
                      |                   | 7  
                      |                   | 8  
                      |                   | 9  
                      |                   | 10 

   a) If \( n \) is the number of items being searched and we had a perfect hash function, what is the average and worst case big-oh notation for a search?
   - average case \( O(\quad) \)
   - worst case \( O(\quad) \)

4. Unfortunately, perfect hash functions are a rarity, so in general many target values might get mapped to the same hash-table index, called a **collision**.

Collisions are handled by two approaches:

- **open-address** with some rehashing strategy: Each hash table home address holds at most one target value. The first target value hashed to a specific home address is stored there. Later targets getting hashed to that home address get rehashed to a different hash table address. A simple rehashing strategy is **linear probing** where the hash table is scanned circularly from the home address until an empty hash table address is found.

- **chaining, closed-address, or external chaining**: all target values hashed to the same home address are stored in a data structure (called a **bucket**) at that index (typically a linked list, but a BST or AVL-tree could also be used). Thus, the hash table is an array of linked list (or whatever data structure is being used for the buckets)
5. Consider the following examples using open-address approach with a simple rehashing strategy of linear probing where the hash table is scanned circularly from the home address until an empty hash table address is found.

<table>
<thead>
<tr>
<th>Set of Keys</th>
<th>Hash function</th>
<th>Hash Table Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>hash(John Doe) = 6</td>
<td>0</td>
</tr>
<tr>
<td>Philip East</td>
<td>hash(Philip East) = 3</td>
<td>1</td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>hash(Mark Fienup) = 5</td>
<td>2</td>
</tr>
<tr>
<td>Ben Schafer</td>
<td>hash(Ben Schafer) = 8</td>
<td>3</td>
</tr>
<tr>
<td>Paul Gray (3-5917)</td>
<td>hash(Paul Gray) = 3</td>
<td>4</td>
</tr>
<tr>
<td>Sarah Diesburg (3-3-7395)</td>
<td>hash(Sarah Diesburg) = 3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Philip East 3-2939</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mark Fienup 3-5918</td>
</tr>
<tr>
<td></td>
<td></td>
<td>John Doe 3-4567</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ben Schafer 3-2187</td>
</tr>
</tbody>
</table>

a) Assuming open-address with linear probing where would Paul Gray and then Sarah Diesburg be placed?

Common rehashing strategies include the following.

<table>
<thead>
<tr>
<th>Rehash Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear probing</td>
<td>Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)</td>
</tr>
<tr>
<td>quadratic probing</td>
<td>Check the square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)^2 + (rehash attempt #))/2) % (hash table size), where the hash table size is a power of 2</td>
</tr>
<tr>
<td>double hashing</td>
<td>Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) * offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size</td>
</tr>
</tbody>
</table>

b) Assume quadratic probing, insert “Paul Gray” and “Sarah Diesburg” into the hash table.

<table>
<thead>
<tr>
<th>Set of Keys</th>
<th>Hash function</th>
<th>Hash Table Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>hash(John Doe) = 6</td>
<td>0</td>
</tr>
<tr>
<td>Philip East</td>
<td>hash(Philip East) = 3</td>
<td>1</td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>hash(Mark Fienup) = 5</td>
<td>2</td>
</tr>
<tr>
<td>Ben Schafer</td>
<td>hash(Ben Schafer) = 0</td>
<td>3</td>
</tr>
<tr>
<td>Paul Gray (3-5917)</td>
<td>hash(Paul Gray) = 3</td>
<td>4</td>
</tr>
<tr>
<td>Sarah Diesburg (3-7395)</td>
<td>hash(Sarah Diesburg) = 3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ben Schafer 3-2187</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Philip East 3-2939</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mark Fienup 3-5918</td>
</tr>
<tr>
<td></td>
<td></td>
<td>John Doe 3-4567</td>
</tr>
</tbody>
</table>
c) Assume double hashing, insert “Paul Gray” and “Sarah Diesburg” into the hash table.

<table>
<thead>
<tr>
<th>Set of Keys</th>
<th>Hash function</th>
<th>Hash Table Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>hash(John Doe) = 6</td>
<td>0  Ben Schafer 3-2187</td>
</tr>
<tr>
<td>Philip East</td>
<td>hash(Philip East) = 3</td>
<td>1  Philip East 3-2939</td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>hash(Mark Fienup) = 5</td>
<td>2  Mark Fienup 3-5918</td>
</tr>
<tr>
<td>Ben Schafer</td>
<td>hash(Ben Schafer) = 0</td>
<td>3  John Doe 3-4567</td>
</tr>
<tr>
<td>Paul Gray</td>
<td>hash(Paul Gray) = 3</td>
<td>4  rehash offset(Paul Gray) = 1</td>
</tr>
<tr>
<td>(3-5917)</td>
<td>rehash offset(Paul Gray) = 1</td>
<td>5  Mark Fienup 3-5918</td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td>hash(Sarah Diesburg) = 3</td>
<td>6  John Doe 3-4567</td>
</tr>
<tr>
<td>(3-7395)</td>
<td>rehash offset(Sarah Diesburg) = 3</td>
<td>7</td>
</tr>
</tbody>
</table>

d) For the above double-hashing example, what would be the sequence of hashing and rehashing addresses tried for Sarah Diesburg if the table was full? For the above example, \((\text{home address} + (\text{rehash attempt} \#) \times \text{offset}) \mod \text{hash table size})\) would be: \((3 + (\text{rehash attempt} \#) \times 3) \mod 8\)

<table>
<thead>
<tr>
<th>Rehash Attempt #</th>
<th>0</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>Address</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>

---

e) Indicate whether each of the following rehashing strategies suffer from primary or secondary clustering.

- **primary clustering** - keys mapped to a home address follow the same rehash pattern
- **secondary clustering** - rehash patterns from initially different home addresses merge together

<table>
<thead>
<tr>
<th>Rehash Strategy</th>
<th>Description</th>
<th>Suffers from:</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear probing</td>
<td>Check next spot (counting circularly) for the first available slot, i.e., ((\text{home address} + (\text{rehash attempt} #)) \mod \text{hash table size}))</td>
<td>primary clustering</td>
</tr>
<tr>
<td>quadratic probing</td>
<td>Check a square of the attempt-number away for an available slot, i.e., ((\text{home address} + ((\text{rehash attempt} #)^2 + (\text{rehash attempt} #))/2) \mod \text{hash table size})), where the hash table size is a power of 2</td>
<td>primary clustering</td>
</tr>
<tr>
<td>double hashing</td>
<td>Use the target key to determine an offset amount to be used each attempt, i.e., ((\text{home address} + (\text{rehash attempt} #) \times \text{offset}) \mod \text{hash table size})), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size</td>
<td>primary clustering</td>
</tr>
</tbody>
</table>

6. Let \(\lambda\) be the load factor (# item/hash table size). The average probes with linear probing for insertion or unsuccessful search is: \(\left(\frac{1}{2}\right)\left(1 + \left(\frac{1}{(1-\lambda)^2}\right)\right)\). The average for successful search is: \(\left(\frac{1}{2}\right)\left(1 + \left(\frac{1}{(1-\lambda)}\right)\right)\).

a) Why is an unsuccessful search worse than a successful search?
The average probes with **quadratic probing** for insertion or unsuccessful search is: \( \left( \frac{1}{1 - \lambda^2} \right) - \lambda - \log_e(1 - \lambda) \)

The average probes with quadratic probing for successful search is: \( 1 - \left( \frac{1}{2} \right) - \log_e(1 - \lambda) \)

Consider the following table containing the average number probes for various load factors:

<table>
<thead>
<tr>
<th>Probing Type</th>
<th>Search outcome</th>
<th>Load Factor, ( \lambda )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>unsuccessful</td>
<td>0.25, 0.5, 0.67, 0.8, 0.99</td>
</tr>
<tr>
<td>Probing</td>
<td>successful</td>
<td>1.39, 2.50, 5.09, 13.00, 5600.50</td>
</tr>
<tr>
<td>Quadratic</td>
<td>unsuccessful</td>
<td>1.17, 1.50, 2.02, 3.00, 50.50</td>
</tr>
<tr>
<td>Probing</td>
<td>successful</td>
<td>1.37, 2.19, 3.47, 5.81, 103.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.16, 1.44, 1.77, 2.21, 5.11</td>
</tr>
</tbody>
</table>

b) Why do you suppose the "general rule of thumb" in hashing tries to keep the load factor between 0.5 and 0.67?

---

7. Allowing deletions from an open-address hash table complicates the implementation. Assuming linear probing we might have the following:

<table>
<thead>
<tr>
<th>Set of Keys</th>
<th>Hash function</th>
<th>Hash Table Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>hash(John Doe) = 6</td>
<td>0</td>
</tr>
<tr>
<td>Philip East</td>
<td>hash(Philip East) = 3</td>
<td>1</td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>hash(Mark Fienup) = 5</td>
<td>2</td>
</tr>
<tr>
<td>Ben Schafer</td>
<td>hash(Ben Schafer) = 8</td>
<td>3</td>
</tr>
<tr>
<td>Paul Gray</td>
<td>hash(Paul Gray) = 3</td>
<td>4</td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td>hash(Sarah Diesburg) = 4</td>
<td>5</td>
</tr>
</tbody>
</table>

|                   | Philip East 3-2939    | 6               |
|                   | Paul Gray 3-5917      | 7               |
|                   | Mark Fienup 3-5918    | 8               |
|                   | John Doe 3-4567       | 9               |
|                   | Sarah Diesburg 3-7395 | 10              |
|                   | Ben Schafer 3-2187    |                |

a) If "Mark Fienup" is deleted, how will we find Sarah Diesburg?

b) How might we fix this problem?
1. The Map/Dictionary abstract data type (ADT) stores key-value pairs. The key is used to look up the data value.

<table>
<thead>
<tr>
<th>Method call</th>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d = ListDict()</code></td>
<td><code>_init_(self)</code></td>
<td>Constructs an empty dictionary</td>
</tr>
<tr>
<td><code>d[&quot;Name&quot;] = &quot;Bob&quot;</code></td>
<td><code>_setItem_(self, key, value)</code></td>
<td>Inserts a key-value entry if key does not exist or replaces the old value with value if key exists.</td>
</tr>
<tr>
<td><code>temp = d[&quot;Name&quot;]</code></td>
<td><code>_getItem_(self, key)</code></td>
<td>Given a key return it value or None if key is not in the dictionary</td>
</tr>
<tr>
<td><code>del d[&quot;Name&quot;]</code></td>
<td><code>_delItem_(self, key)</code></td>
<td>Removes the entry associated with key</td>
</tr>
<tr>
<td><code>if &quot;Name&quot; in d:</code></td>
<td><code>_contains_(self, key)</code></td>
<td>Return True if key is in the dictionary; return False otherwise</td>
</tr>
<tr>
<td><code>for k in d:</code></td>
<td><code>_iter_(self)</code></td>
<td>Iterates over the keys in the dictionary</td>
</tr>
<tr>
<td><code>len(d)</code></td>
<td><code>_len_(self)</code></td>
<td>Returns the number of items in the dictionary</td>
</tr>
<tr>
<td><code>str(d)</code></td>
<td><code>_str_(self)</code></td>
<td>Returns a string representation of the dictionary</td>
</tr>
</tbody>
</table>

---

class Entry(object):
    """A key/value pair."""
    
    def _init_(self, key, value):
        self._key = key
        self._value = value

    def getKey(self):
        return self._key

    def getValue(self):
        return self._value

    def setValue(self, newvalue):
        self._value = newvalue

    def _eq_(self, other):
        if not isinstance(other, Entry):
            return False
        return self._key == other._key

    def _str_(self):
        return str(self._key) + ":" + str(self._value)

from entry import Entry

class ListDict(object):
    """Dictionary implemented with a Python list."""
    
    def _init_(self):
        self._table = []

    def _getItem_(self, key):
        """Returns the value associated with key or returns None if key does not exist."""
        entry = Entry(key, None)
        try:
            index = self._table.index(entry)
            return self._table[index].getValue()
        except:
            return None

    def _delItem_(self, key):
        """Removes the entry associated with key."""
        entry = Entry(key, None)
        try:
            index = self._table.index(entry)
            self._table.pop(index)
        except:
            return

    def _str_(self):
        """Returns string repr. of the dictionary"""
        resultStr = "[
        for item in self._table:
            resultStr = resultStr + " " + str(item)
        return resultStr + "]"

    def _iter_(self):
        """Iterates over keys of the dictionary"""
        for item in self._table:
            yield item.getKey()
        raise StopIteration

---

a) Complete the code for the _contains_ method.

    def _contains_(self, key):

b) Complete the code for the _setItem_ method.

    def _setItem_(self, key, value):

2. Dictionary implementation using hashing with chaining -- an UnorderedList object at each slot in the hash table.

```python
from entry import Entry
from unordered_linked_list import UnorderedList

class ChainingDict(object):
    """Dictionary implemented using hashing with chaining."""
    def __init__(self, capacity=8):
        self.capacity = capacity
        self._table = []
        for index in range(self.capacity):
            self._table.append(UnorderedList())
        self._size = 0
        self._index = None

def _contains_(self, key):
    """Returns True if key is in the dictionary or False otherwise."""
    self.index = abs(hash(key)) % self.capacity
    entry = Entry(key, None)
    return self._table[self._index].search(entry)

def _getitem_(self, key):
    """Returns the value associated with key or returns None if key does not exist."""
    if key in self:
        entry = Entry(key, None)
        entry - self._table[self._index].remove(entry)
        self._table[self._index].add(entry)
        return entry.getValue()
    else:
        return None

def _delitem_(self, key):
    """Removes the entry associated with key."""
    if key in self:
        entry = Entry(key, None)
        entry - self._table[self._index].remove(entry)
        self._size -= 1

def _setitem_(self, key, value):
    """Inserts an entry with key/value if key does not exist or replaces the existing value with value if key exists."""
    entry = Entry(key, value)
    if key in self:
        entry = self._table[self._index].remove(entry)
        entry.setValue(value)
    else:
        self._size += 1
        self._table[self._index].add(entry)

def __len__(self):
    return self._size

def __str__(self):
    result = "HashDict: capacity = " + str(self.capacity) + ", load factor = " + str(len(self) / self.capacity)
    for i in range(self.capacity):
        result += "\nRow " + str(i)+": " + str(self._table[i])
    return result

def __iter__(self):
    """Iterates over the keys of the dictionary""
```

a) In `__getitem__`, why is the `entry = Entry(key, None)` object created?

b) In `__getitem__`, where does `self._index` receive its value?

c) What single modification was needed to the `UnorderedList`'s `remove` method?

d) Complete the `__iter__` method.
Objective: To experiment with searching and get a feel for the performance of hashing.

To start the lab: Download and unzip the file lab7.zip

Part A:

a) Open and run the timeLinearSearch.py program that times the LinearSearch algorithm imported from LinearSearch.py. Observe that it creates a list, evenList, that holds 10,000 sorted, even values (e.g., evenList = [0, 2, 4, 6, 8, ..., 19996, 19998]). It then times the searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. How long does it take to linear search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999?

b) Open and run the timeBinarySearch.py program that times the binarySearch algorithm imported from binarySearch.py. How long does it take to binary search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999?

c) Open and run the timeListDictSearch.py program that times the ListDict dictionary ADT in list_dictionary.py. The ListDict implementation uses a single Python list for storing dictionary entries. The timeListDictSearch.py program adds the 10,000 even values (i.e., 0, 2, 4, 6, 8, ..., 19996, 19998) to a ListDict object, and then times the searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 in the ListDict?

d) Open and run the timeChainingDictSearch.py program that times the ChainingDict dictionary ADT in chaining_dictionary.py. The timeChainingDictSearch.py program adds the 10,000 even values (i.e., 0, 2, 4, 6, 8, ..., 19996, 19998) to a ChainingDict with 16,384 slots (i.e., load factor of 0.61), and then times the searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 in the ChainingDict?

e) Explain the relative performance results of searching using linear search, binary search, a ListDict, and ChainingDict.

f) The Python for loop allows traversal of built-in data structures (strings, lists, tuple, etc) by an iterator. To accomplish this with our data structures we need to include an __iter__ method (e.g., ListDict class from Lecture 15 at http://www.cs.uni.edu/~fienup/cs1520f16/lectures/lec15_questions.pdf). In general an __iter__ method, must loop down the data structure and yield each item in the data structure. When done, the __iter__ needs to raise StopIteration. See the end of UnorderedList and ListDict classes for examples of their __iter__ methods. Complete the __iter__ code for the ChainingDict and OpenAddrHashDict classes.

After you have completed the above timings, questions and code, raise your hand and explain your answers.
Part B:

a) Open and run the `timeOpenAddrHashDictSearch.py` program that times the `OpenAddrHashDict` dictionary ADT in `open_addr_hash_dictionary.py`. The `timeOpenAddrHashDictSearch.py` program adds the 10,000 even values (i.e., 0, 2, 4, 6, 8, ..., 19996, 19998) to an `OpenAddrHashDict` with 16,384 (2^14) slots (i.e., load factor of 0.61) using linear probing, and then times the searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 in the `OpenAddrHashDict`?

b) Place the even values (i.e., 0, 2, 4, 6, 8, ..., 19996, 19998) in the hash table below. Value 0 is stored at home address 0, value 2 is stored at home address 2, ..., value 16,382 is stored at home address 16,382, but values 16,384 to 19,998 will have collisions. Now, think about the number of probes needed to searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999. Why does the above timing of searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 take so long with a load factor of only 0.61?

```
<table>
<thead>
<tr>
<th>self_table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>16,370</td>
</tr>
<tr>
<td>16,371</td>
</tr>
<tr>
<td>16,372</td>
</tr>
<tr>
<td>16,373</td>
</tr>
</tbody>
</table>
```

c) Experiment with changing the load factor of the HashTable by increasing the hash table size to 32,768 (2**15) for a load factor of 0.31, and 65,536 for a load factor of 0.15. Completing the following table:

<table>
<thead>
<tr>
<th>Linear Probing</th>
<th>Hash Table Size (Load Factor)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16,384 (0.61)</td>
</tr>
<tr>
<td>Execution time with 10,000 items in hash table (seconds)</td>
<td></td>
</tr>
</tbody>
</table>

d) In `timeOpenAddrHashDictSearch.py` modify the construction of `evenHashTable` so it uses quadratic probing instead of linear probing (i.e., `evenHashTable = OpenAddrHashTable(2**14, hash, False)`). Completing the following table:

<table>
<thead>
<tr>
<th>Quadratic Probing</th>
<th>Hash Table Size (Load Factor)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16,384 (0.61)</td>
</tr>
<tr>
<td>Execution time with 10,000 items in hash table (seconds)</td>
<td></td>
</tr>
</tbody>
</table>

e) Explain why quadratic probing performs better than linear probing.

After you have performed the timings and answered the questions, raise your hand and explain your answers.
class OpenAddrHashDict(object):
    EMPTY = None  # class variables shared by all objects of the class
    DELETED = True

    def __init__(self, capacity = 8, hashFunction = hash,
                 linear = True):
        self._table = [OpenAddrHashDict.EMPTY] * capacity
        self._size = 0
        self._hash = hashFunction
        self._homeIndex = -1
        self._actualIndex = -1
        self._linear = linear
        self._probeCount = 0

    def __getitem__(self, key):
        """Returns the value associated with key or returns None if key does not exist.""
        if key in self:
            return self._table[self._actualIndex].getValue()
        else:
            return None

    def __delitem__(self, key):
        """Removes the entry associated with key.""
        if key in self:
            self._table[self._actualIndex] = OpenAddrHashDict.DELETED
            self._size -= 1

    def __setitem__(self, key, value):
        """Inserts an entry with key/value if key does not exist or replaces the existing value with value if key exists.""
        entry = Entry(key, value)
        if key in self:
            self._table[self._actualIndex] = entry
        else:
            self._table[self._actualIndex] = entry
            self._size += 1

    def __contains__(self, key):
        """Return True if key is in the dictionary; return False otherwise""
        entry = Entry(key, None)
        self._probeCount = 0
        # Get the home index
        self._homeIndex = abs(self._hash(key)) % len(self._table)
        rehashAttempt = 0
        index = self._homeIndex
        # Stop searching when an empty cell is encountered
        while rehashAttempt < len(self._table):
            self._probeCount += 1
            if self._table[index] == OpenAddrHashDict.EMPTY:
                self._actualIndex = index
                return False  # An empty cell is found, so key not found
            elif self._table[index] == entry:
                self._actualIndex = index
                return True
            # Calculate the index and wrap around to first position if necessary
            if self._linear:
                index = (self._homeIndex + rehashAttempt) % len(self._table)
            else:  # Quadratic probing
                index = (self._homeIndex + (rehashAttempt ** 2 + rehashAttempt)) // 2 % len(self._table)
            rehashAttempt += 1
            # try all the slots in the hash table and did not find key

    def __len__(self):
        return self._size

    def __str__(self):
        resultStr = "[
        for item in self._table:
            if not item in (OpenAddrHashDict.EMPTY, OpenAddrHashDict.DELETED):
                resultStr = resultStr + " " + str(item)
        return resultStr + " ]"

    def __iter__(self):
        """Iterates over the keys of the dictionary""

a) Complete the __iter__ method.
2. All simple sorts consist of two nested loops where:

- the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
- the inner loop's job is to do the work to extend the sorted part's size by one.

Initially, the sorted part is typically empty. The simple sorts differ in how their inner loops perform their job.

Selection sort is an example of a simple sort. Selection sort's inner loop scans the unsorted part of the list to find the maximum item. The maximum item in the unsorted part is then exchanged with the last unsorted item to extend the sorted part by one item.

At the start of the first iteration of the outer loop, initial list is completely unsorted:

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Empty Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>25 35 20 40 90 60 10 50 45</td>
<td></td>
</tr>
</tbody>
</table>

The inner loop scans the unsorted part and determines that the index of the maximum item, \( maxIndex = 4 \).

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>25 35 20 40 90 60 10 50 45</td>
<td></td>
</tr>
</tbody>
</table>

\[ maxIndex = 4 \quad lastUnsortedIndex = 8 \]

After the inner loop (but still inside the outer loop), the item at \( maxIndex \) is exchanged with the item at \( lastUnsortedIndex \). Thus, extending the Sorted Part of the list by one item.

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>25 35 20 45 60 10 50 90</td>
<td></td>
</tr>
</tbody>
</table>

\[ maxIndex = 4 \quad lastUnsortedIndex = 8 \]

a) Write the code for the outer loop

b) Write the code for the inner loop to scan the unsorted part of the list to determine the index of the maximum item

c) Write the code to exchange the list items at positions \( maxIndex \) and \( lastUnsortedIndex \).

d) What is the big-oh notation for selection sort?
3. Bubble sort is another example of a simple sort. Bubble sort’s inner loop scans the unsorted part of the list comparing adjacent items. If it finds adjacent items out of order, then it exchanges them. This causes the largest item to "bubble" up to the "top" of the unsorted part of the list.

At the start of the first iteration of the outer loop, initial list is completely unsorted:

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Empty Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>25 35 20 40 90 60 10 50 45</td>
<td></td>
</tr>
</tbody>
</table>

The inner loop scans the unsorted part by comparing adjacent items and exchanging them if out of order.

Unsorted Part | Sorted Part |
---|---|
| 0 1 2 3 4 5 6 7 8 |
| 25 35 20 40 90 60 10 50 45 |

- in order, so don't exchange
- out of order, so exchange

Unsorted Part | Sorted Part |
---|---|
| 0 1 2 3 4 5 6 7 8 |
| 25 20 35 40 90 60 10 50 45 |

- in order, so don't exchange
- in order, so don't exchange
- out of order, so exchange

Unsorted Part | Sorted Part |
---|---|
| 0 1 2 3 4 5 6 7 8 |
| 25 20 35 40 60 90 10 50 45 |

- out of order, so exchange

Unsorted Part | Sorted Part |
---|---|
| 0 1 2 3 4 5 6 7 8 |
| 25 20 35 40 60 10 90 50 45 |

- out of order, so exchange

Unsorted Part | Sorted Part |
---|---|
| 0 1 2 3 4 5 6 7 8 |
| 25 20 35 40 60 10 50 90 45 |

- out of order, so exchange

Unsorted Part | Sorted Part |
---|---|
| 0 1 2 3 4 5 6 7 8 |
| 25 20 35 40 60 10 50 45 90 |

After the inner loop (but still inside the outer loop), there is nothing to do since the exchanges occurred inside the inner loop.

a) What would be the worst-case big-oh of bubble sort?

b) What would be true if we scanned the unsorted part and didn’t need to do any exchanges?
4. Another simple sort is called insertion sort. Recall that in a simple sort:
  • the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
  • the inner loop's job is to do the work to extend the sorted part's size by one.

After several iterations of insertion sort's outer loop, a list might look like:

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>10 20 35 40 45 60 25</td>
<td>50 90 ● ● ●</td>
</tr>
</tbody>
</table>

In insertion sort the inner-loop takes the "first unsorted item" (25 at index 6 in the above example) and "inserts" it into the sorted part of the list "at the correct spot." After 25 is inserted into the sorted part, the list would look like:

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>10 20 25 35 40 45 60 50 90 ● ● ●</td>
<td></td>
</tr>
</tbody>
</table>

Code for insertion is given below:

```python
def insertionSort(myList):
    """Rearranges the items in myList so they are in ascending order"""

    for firstUnsortedIndex in range(1, len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1

        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1

        # Insert the itemToInsert at the correct spot
        myList[testIndex + 1] = itemToInsert
```

a) What is the purpose of the `testIndex >= 0` while-loop comparison?

b) What initial arrangement of items causes the is the overall worst-case performance of insertion sort?

c) What is the worst-case \( O() \) notation for the number of item moves?

d) What is the worst-case \( O() \) notation for the number of item comparisons?

e) What initial arrangement of items causes the is the overall best-case performance of insertion sort?

f) What is the best-case \( O() \) notation for insertion sort?
1. So far, we have looked at simple sorts consisting of nested loops. The \( n(n-1)/2 \) iterations are \( O(n^2) \). Consider using a min-heap to sort a list. (methods: BinHeap\((), \) insert\((), \) delMin\((), \) isEmpty\((), \) size\())

a) If we insert all of the list elements into a min-heap, what would we easily be able to determine?

- **General idea of Heap sort:**
  - myList: unsorted list with \( n \) items
  - heap with \( n \) items
  - myList: sorted list with \( n \) items

b) What is the overall \( O() \) for heap sort?

2. Another way to do better than the simple sorts is to employ divide-and-conquer (e.g., Merge sort and Quick Sort). Recall the idea of **Divide-and-Conquer** algorithms. Solve a problem by:
  - dividing problem into smaller problem(s) of the same kind
  - solving the smaller problem(s) recursively
  - use the solution(s) to the smaller problem(s) to solve the original problem
In general, a problem can be solved recursively if it can be broken down into smaller problems that are identical in structure to the original problem.

a) What determines the "size" of a sorting problem?

b) How might we break the original problem down into smaller problems that are identical?

c) What base case(s) (i.e., trivial, non-recursive case(s)) might we encounter with recursive sorts?

d) How do you combine the answers to the smaller problems to solve the original sorting problem?

e) Consider why a recursive sort might be more efficient. Assume that I had a simple \( n^2 \) sorting algorithm with \( n = 100 \), then there is roughly \( 100^2 / 2 \) or 5,000 amount of work. Suppose I split the problem down into two smaller sorting problems of size 50.
  - If I run the \( n^2 \) algorithm on both smaller problems of size 50, then what would be the approximate amount of work?
  - If I further solve the problems of size 50 by splitting each of them into two problems of size 25, then what would be the approximate amount of work?
3. The general idea merge sort is as follows. Assume \( n \) items to sort.
- Split the unsorted part in half to get two smaller sorting problems of about equal size = \( n/2 \)
- Solve both smaller problems recursively using merge sort
- "Merge" the solutions to the smaller problems together to solve the original sorting problem of size \( n \)

a) Fill in the merged Sorted Part in the diagram.

b) Describe how you filled in the sorted part in the above example?

4. Merge sort is substantially faster than the simple sorts. Let’s analyze the number of comparisons and moves of merge sort. Assume \( n \) items to sort.

a) On each level of the above diagram write the WORST-CASE number of comparisons and moves for that level.

b) What is the WORST-CASE total number of comparisons and moves for the whole algorithm (i.e., add all levels)?

c) What is the big-oh for worst-case?
5. *Quick sort* general idea is as follows.
- Select a "random" item in the unsorted part as the *pivot*
- Rearrange (*partitioning*) the unsorted items such that:
  - Quick sort the unsorted part to the left of the pivot
  - Quick sort the unsorted part to the right of the pivot

a) Given the following *partition* function which returns the index of the pivot after this rearrangement, complete the recursive *quicksortHelper* function.

```
def partition(lyst, left, right):
    # Find the pivot and exchange it with the last item
    middle = (left + right) // 2
    pivot = lyst[middle]
    lyst[middle] = lyst[right]
    lyst[right] = pivot
    # Set boundary point to first position
    boundary = left
    # Move items less than pivot to the left
    for index in range(left, right):
        if lyst[index] < pivot:
            temp = lyst[index]
            lyst[index] = lyst[boundary]
            lyst[boundary] = temp
            boundary += 1
    # Exchange the pivot item and the boundary item
    temp = lyst[boundary]
    lyst[boundary] = lyst[right]
    lyst[right] = temp
    return boundary
```

def quicksort(lyst):
    quicksortHelper(lyst, 0, len(lyst) - 1)

def quicksortHelper(lyst, left, right):
```
b) For the list below, trace the first call to *partition* and determine the resulting list, and value returned.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>26</td>
<td>93</td>
<td>17</td>
<td>50</td>
<td>31</td>
<td>44</td>
<td>55</td>
<td>20</td>
</tr>
</tbody>
</table>

left: | right: | index: | boundary: | pivot:
---|---|---|---|---
0 | 8 | | | 

b) What initial arrangement of the list would cause *partition* to perform the most amount of work?

c) Let "n" be the number of items between left and right. What is the worst-case $O(\ )$ for *partition*?

d) What would be the overall, worst-case $O(\ )$ for *Quick Sort*?
e) Ideally, the pivot item splits the list into two equal size problems. What would be the big-oh for Quick Sort in the best case?

f) What would be the big-oh for Quick Sort in the average case?

g) The textbook’s partition code (Listing 5.15 on page 225) selects the first item in the list as the pivot item. However, the above partition code selects the middle item of the list to be the pivot. What advantage does selecting the middle item as the pivot have over selecting the first item as the pivot?
Objectives: You will gain experience:
- get a feel for simple sorts: selection, bubble, and insertion sorts
- get a feel for advanced sorts: heap, quick, and merge sorts

To start the lab: Download and unzip the file: http://www.cs.uni.edu/~fienup/cs1520fl6/labs/lab8.zip
The lab8.zip file you downloaded and extracted contains the following sorting algorithms which all sort in ascending order (i.e., from smallest to largest):
- bubbleSort.py - bubble sort code which does not check if it can stop early
- bubbleSortB.py - bubble sort code which stops early if no swapping is needed during a scan of the unsorted part
- insertionSort.py - the insertion sort
- selectionSort.py - the selection sort code we developed in class

Each program runs the sorting algorithm several time with different initial orderings of 10,000 list items. The initial orderings of items are: descending order, ascending order, random order 1, and random order 2 again to check for consistence. Complete the following timings by running the each program.

<table>
<thead>
<tr>
<th>Type of sorting algorithm</th>
<th>Initial Ordering of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Descending</td>
</tr>
<tr>
<td>bubbleSort.py</td>
<td></td>
</tr>
<tr>
<td>bubbleSortB.py</td>
<td></td>
</tr>
<tr>
<td>insertionSort.py</td>
<td></td>
</tr>
<tr>
<td>selectionSort.py</td>
<td></td>
</tr>
</tbody>
</table>

Study the code and answer the following questions about the sorting algorithms:

a) Why does the bubbleSort algorithm take less time on the ascending ordered list than the descending ordered list?

b) Why does the bubbleSortB algorithm take A LOT less time on the ascending ordered list than the descending ordered list?

c) Why does the insertionSort algorithm take A LOT less time on the ascending ordered list than the descending ordered list?

d) Why does the insertionSort algorithm take less time on the descending ordered list than the bubbleSort algorithm on the descending ordered list?
Data Structures (CS 1520)  Lab 8  Name:

e) Why does the selectionSort algorithm take less time on the descending ordered list than the insertionSort algorithm on the descending ordered list?

After you have answered the above questions, raise your hand and explain your answers.

Part B:

a) Complete the heap sort function in lab8/heapSort.py which contains the template for the heap sort algorithm discussed in class. Recall the steps of the algorithm:

```
myList | unsorted list with n items

Steps:  
1. Create an empty heap
2. Insert all n list items into heap
3. delMin heap items back to list in sorted order

myList | sorted list with n items
```

b) Time the heap sorting algorithm using lab8/timeHeapSort.py on 100,000 random items, 200,000 random items, and 400,000 random items.

<table>
<thead>
<tr>
<th># Items</th>
<th>Your Heap Sort Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td></td>
</tr>
</tbody>
</table>

c) Explain the $O(\cdot)$ for your heap sort algorithm?
The general idea merge sort is as follows. Assume “n” items to sort.
- Split the unsorted part in half to get two smaller sorting problems of about n/2
- Solve both smaller problems recursively using merge sort
- “Merge” the solution to the smaller problems together to solve the original sorting problem of size n

The textbook’s merge sort is in mergesort.py. Use the timeMergeSort.py program to run merge sort on a list of random items. Complete the following timing table:

<table>
<thead>
<tr>
<th>Random # Items</th>
<th>Textbook’s Merge Sort Timings</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td></td>
</tr>
</tbody>
</table>

e) Recall the general idea of Quick sort is as follows. Assume “n” items to sort.
- Select a “random” item in the unsorted part as the pivot
- Rearrange (called partitioning) the unsorted items such that:

Pivot Index

<table>
<thead>
<tr>
<th>All items &lt; to Pivot</th>
<th>Pivot Item</th>
<th>All items &gt;= to Pivot</th>
</tr>
</thead>
</table>

- Quick sort the unsorted part to the left of the pivot
- Quick sort the unsorted part to the right of the pivot

The lecture 17 quick sort is in quicksort.py. Use the timeQuickSort.py program to run quick sort on a list of random items. Complete the following timings to get a feel for the “speed” of quicksort.

<table>
<thead>
<tr>
<th># Items</th>
<th>Lecture 17 Quick Sort Timings</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td></td>
</tr>
</tbody>
</table>

All three advanced sorting algorithms are O(nlogn) on initially random data. Why do you suppose quick sort is the fastest advanced sort on random items?

After you have completed the above times and answered the above question, raise your hand and explain your answers.
Part C: EXTRA CREDIT

a) Write (pencil-and-paper below) a variation of bubble sort that:
   • sorts in descending order (largest to smallest)
   • builds the sorted part on the left-hand side of the list, i.e.,

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   Inner loop scans from right to left across the unsorted part swapping adjacent items that are "out of order"

(Your code does NOT need to stop early if a scan of the unsorted part has no swaps)

def bubbleSortC(myList):

b) Implement and test your bubbleSortC code.
1. Consider the parse tree for \((9 + (5 * 3)) / (8 - 4)\):

   ![Parse Tree Diagram]

   a) Identify the following items in the above tree:
   - node containing "*"
   - edge from node containing "." to node containing "8"
   - root node
   - children of the node containing "+"
   - parent of the node containing "3"
   - siblings of the node containing "*"
   - leaf nodes of the tree
   - subtree whose root is node contains "4"
   - path from node containing "+" to node containing "5"
   - branch from root node to "3"

   b) Mark the levels of the tree (level is the number of edges on the path from the root)

   c) What is the height (max. level) of the tree?

2. In Python an easy way to implement a tree is as a list of lists where a tree look like:

   ```
   [ ["node value", remaining items are subtrees for the node each implemented as a list of lists]
   ```

   Complete the list-of-lists representation look like for the above parse tree:

   ```
   [ ["/", ["+", ["5", ["*", ["9", ["_", None], None], ["3", ["_", None], None]], None]], ["-", ["8", None], ["4", None]]]
   ```

3. Consider a "linked" representations of a BinaryTree. For the expression \(((4 + 5) * 7)\), the binary tree would be:

   ```
   class BinaryTree:
   def __init__(self, rootObj):
       self.key = rootObj
       self.leftChild = None
       self.rightChild = None
   ```
import operator
class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None

def insertLeft(self, newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.left = self.leftChild
        self.leftChild = t

def insertRight(self, newNode):
    if self.rightChild == None:
        self.rightChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.right = self.rightChild
        self.rightChild = t

def isLeaf(self):
    return (not self.leftChild and
            not self.rightChild)

def getRightChild(self):
    return self.rightChild

def getLeftChild(self):
    return self.leftChild

def setRootVal(self, obj):
    self.key = obj

def getRootVal(self):
    return self.key

def inorder(self):
    if self.leftChild:
        self.leftChild.inorder()
    print(self.key)
    if self.rightChild:
        self.rightChild.inorder()

def postorder(self):
    if self.leftChild:
        self.leftChild.postorder()
    if self.rightChild:
        self.rightChild.postorder()
    print(self.key)

def preorder(self):
    print(self.key)
    if self.leftChild:
        self.leftChild.preorder()
    if self.rightChild:
        self.rightChild.preorder()

def printexp(self):
    if self.leftChild:
        print('(', end=' ')
        self.leftChild.printexp()
    print(self.key, end=' ')
    if self.rightChild:
        self.rightChild.printexp()
    print(')', end=' ')

def postOrderEval(self):
    operes = {'+':operator.add, '-':operator.sub,
              '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if self.leftChild:
        res1 = self.leftChild.postOrderEval()
    if self.rightChild:
        res2 = self.rightChild.postOrderEval()
    if res1 and res2:
        return operes[self.key](res1, res2)
    else:
        return self.key

def height(tree):
    if tree == None:
        return -1
    else:
        return 1 +
        max(height(tree.leftChild),
            height(tree.rightChild))
b) If `myTree` is the `BinaryTree` object for the expression: `(4 + 5) * 7`, what gets printed by calls to:

<table>
<thead>
<tr>
<th><code>myTree.inorder()</code></th>
<th><code>myTree.preorder()</code></th>
<th><code>myTree.postorder()</code></th>
<th><code>inorder(myTree)</code></th>
</tr>
</thead>
</table>


c) If `myTree` is the `BinaryTree` object for the expression: `(4 + 5) * 7`, what gets printed by a call to `myTree.printexp()`?

d) If `myTree` is the `BinaryTree` object for the expression: `(4 + 5) * 7`, what gets printed by a call to `myTree.postordereval()`?

c) From an class/Abstract Data Type (ADT) point of view, why are the external versions of the methods “better”?

f) Write the `height` method for the `BinaryTree` class.

4. Consider the Binary Search Tree (BST). For each node, all values in the left-subtree are < the node and all values in the right-subtree are > the node.

```
      50
     / \
   30   70
  / \   / \
 9   34 58 80
 / \   \   \   \
18  32 47 82 83
```

a. Starting at the root, how would you find the node containing “32”?
b. Starting at the root, when would you discover that “65” is not in the BST?
c. Starting at the root, where would be the “easiest” place to add “65”?
d. Where would we add “5” and “33”?
1. Consider the partial TreeNode class and partial BinarySearchTree class.

```python
class TreeNode:
    def __init__(self, key, val, left=None, right=None, parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

def hasLeftChild(self):
    return self.leftChild

def hasRightChild(self):
    return self.rightChild

def isLeftChild(self):
    return self.parent and \n    self.parent.leftChild == self

def isRightChild(self):
    return self.parent and \n    self.parent.rightChild == self

def isRoot(self):
    return not self.parent

def isLeaf(self):
    return not (self.rightChild or self.leftChild)

def hasAnyChildren(self):
    return self.rightChild or self.leftChild

def hasBothChildren(self):
    return self.rightChild and self.leftChild

def replaceNodeData(self, key, value, lc, rc):
    self.key = key
    self.payload = value
    self.leftChild = lc
    self.rightChild = rc
    if self.hasLeftChild():
        self.leftChild.parent = self
    if self.hasRightChild():
        self.rightChild.parent = self

def __iter__(self):
    if self:
        if self.hasLeftChild():
            for elem in self.leftChild:
                yield elem
        yield self.key
        if self.hasRightChild():
            for elem in self.rightChild:
                yield elem
```

A BinarySearchTree object

- **size**: root is None if size is 0
- **TreeNode objects**

```python
class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()

    def __str__(self):
        """"Returns a string representation of the tree
        rotated 90 degrees counter-clockwise""

        def strHelper(root, level):
            resultStr = ''
            if root:
                resultStr += strHelper(root.rightChild, level+1)
                resultStr += "| " * level
                resultStr += str(root.key) + "\n"
                resultStr += strHelper(root.leftChild, level+1)
            return resultStr

    resultStr = strHelper(self.root, 0)
```

a) How do the BinarySearchTree __iter__ and __str__ methods work?
More partial TreeNode class and partial BinarySearchTree class.

class BinarySearchTree:
    ...
    def _contains_(self, key):
        if self.get(key, self.root):
            return True
        else:
            return False
    def get(self, key):
        if self.root:
            res = self._get(key, self.root)
            if res:
                return res.payload
            else:
                return None
        else:
            return None
    def _get(self, key, currentNode):
        if not currentNode:
            return None
        elif currentNode.key == key:
            return currentNode
        elif key < currentNode.key:
            return self._get(key, currentNode.leftChild)
        else:
            return self._get(key, currentNode.rightChild)
    def __getitem__(self, key):
        return self.get(key)
    def __setitem__(self, k, v):
        self.put(k, v)
    def put(self, key, val):
        if self.root:
            self._put(key, val, self.root)
        else:
            self.root = TreeNode(key, val)
            self.size = self.size + 1
    def _put(self, key, val, currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                ...
            else:
                ...
        else:
            ...
else:
    ...

b) The _get method is the "work horse" of BST search. It recursively walks currentNode down the tree until it finds key or becomes None. In English, what are the base and recursive cases?

c) What is the put method doing?

d) Complete the recursive _put method.

e) Draw the "shape" of the BST after puts of: 50, 60, 30, 70, 90, 40, 65

f) If "n" items are in the BST, what is put's: Best-case O( )? Worst-case O( )? Average-case O( )?
class BinarySearchTree:
    ...
    def delete(self, key):
        if self.size > 1:
            nodeToRemove = self._get(key, self.root)
            if nodeToRemove:
                self.remove(nodeToRemove)
                self.size = self.size - 1
            else:
                raise KeyError('Error, key not in tree')
        elif self.size == 1 and self.root.key == key:
            self.root = None
            self.size = self.size - 1
        else:
            raise KeyError('Error, key not in tree')
    
    def __delitem__(self, key):
        self.delete(key)

    def remove(self, currentNode):
        if currentNode.isLeaf(): # leaf
            if currentNode == currentNode.parent.leftChild:
                currentNode.parent.leftChild = None
            else:
                currentNode.parent.rightChild = None
        elif currentNode.hasBothChildren(): # interior
            succ = currentNode.findSuccessor()
            succ.spliceOut()
            currentNode.key = succ.key
            currentNode.payload = succ.payload

        else: # this node has one child
            if currentNode.hasLeftChild():
                if currentNode.isLeftChild():
                    currentNode.leftChild.parent = currentNode.parent
                    currentNode.parent.leftChild = currentNode.leftChild
                else:
                    currentNode.leftChild.parent = currentNode.parent
                    currentNode.parent.rightChild = currentNode.leftChild
            else:
                currentNode.replaceNodeData(currentNode.leftChild.key,
                                             currentNode.leftChild.payload,
                                             currentNode.leftChild.leftChild,
                                             currentNode.leftChild.rightChild)

            else:
                if currentNode.isLeftChild():
                    currentNode.rightChild.parent = currentNode.parent
                    currentNode.parent.leftChild = currentNode.rightChild
                elif currentNode.isRightChild():
                    currentNode.rightChild.parent = currentNode.parent
                    currentNode.parent.rightChild = currentNode.rightChild
                else:
                    currentNode.replaceNodeData(currentNode.rightChild.key,
                                                 currentNode.rightChild.payload,
                                                 currentNode.rightChild.leftChild,
                                                 currentNode.rightChild.rightChild)

    def search(self, key):
        currentNode = self.root
        while currentNode:
            if currentNode.key == key:
                return currentNode
            elif currentNode.key > key:
                currentNode = currentNode.leftChild
            else:
                currentNode = currentNode.rightChild
        return None

    def findSuccessor(self):
        if self.hasRightChild():
            return self.rightChild.findMin()
        else:
            predecessor = self.parent
            while predecessor and self.key > predecessor.key:
                self = predecessor
                predecessor = predecessor.parent
            return predecessor

    def findMin(self):
        currentNode = self.leftChild
        while currentNode:
            currentNode = currentNode.leftChild
        return currentNode

    def findMax(self):
        currentNode = self.rightChild
        while currentNode:
            currentNode = currentNode.rightChild
        return currentNode

    def __str__(self):
        return f'Size: {self.size}, Root: {self.root}'

if __name__ == '__main__':
    bst = BinarySearchTree()
    bst.add(5)
    bst.add(3)
    bst.add(7)
    print(bst)
3. Yet even more partial TreeNode class and partial BinarySearchTree class.

```python
class TreeNode:
    ...
    def findSuccessor(self):
        succ = None
        if self.hasRightChild():
            succ = self.rightChild.findMin()
        else:
            if self.parent:
                if self.isLeftChild():
                    succ = self.parent
                else:
                    self.parent.rightChild = None
                    succ = self.parent.findSuccessor()
                    self.parent.rightChild = self
        return succ
    
def findMin(self):
        current = self
        while current.hasLeftChild():
            current = current.leftChild
        return current
    
def spliceOut(self):
        if self.isLeaf():
            if self.isLeftChild():
                self.parent.leftChild = None
            else:
                self.parent.rightChild = None
        elif self.hasAnyChildren():
            if self.hasLeftChild():
                if self.isLeftChild():
                    self.parent.leftChild = self.leftChild
                else:
                    self.parent.rightChild = self.leftChild
                    self.leftChild.parent = self.parent
            else:
                if self.isLeftChild():
                    self.parent.leftChild = self.rightChild
                else:
                    self.parent.rightChild = self.rightChild
                    self.rightChild.parent = self.parent
    
```
Objectives: You will gain experience BST performance and implementation

To start the lab: Download and unzip the file: http://www.cs.uni.edu/~fienup/cs1520f16/labs/lab9.zip

Part A: Run the timeBinarySearchTree.py program that:
  • creates a list, evenList, that holds 3,000 sorted, even values (e.g., evenList = [0, 2, 4, 6, 8, ..., 5996, 5998])
  • puts (adds) all the evenList items into an initially empty BinarySearchTree object, bst
  • times the searches (in) bst for target values 0, 1, 2, 3, 4, ..., 5998, 5999 so half of the searches are successful and half are unsuccessful
a) How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 5998, 5999?

b) Explain why these searches take so long. (Hint: consider the shape of the BinarySearchTree bst)

c) Uncomment the “shuffle(evenList)” which randomizes the items in evenList. Now how long does it take to search for target values from 0, 1, 2, 3, 4, ..., 5998, 5999?

d) Explain why these searches take so little time.

e) What is the search time with the timeOpenAddrHashDictSearch.py program? Why is it faster?

After you have answered the above questions, raise your hand and explain your answers.

Part B:  
a) Complete the recursive height method in the BinarySearchTree class (model it after one of the traversals). Uncomment the call to the height method at the end of the timeBinarySearchTree.py program. What is the height of bst if we are shuffling the evenList?

b) What would be the shortest possible height for a binary tree with 3,000 items?

c) Why was it necessary to raise the recursion limit with the statement: sys.setrecursionlimit(10000)?

After you have completed the height method and answered the above questions, raise your hand and explain your answers.
1. Consider the Binary Search Tree (BST):

```
   50
  /   \
30    60
 /  \
9    34  58
|    / \|
18  32 47 53
     /   \
    55
```

a. What would need to be done to delete 32 from the BST?

b. What would need to be done to delete 9 from the BST?

c. What would be the result of deleting 50 from the BST? Hint: One technique when programming is to convert a hard problem into a simpler problem. Deleting a BST node that contains two children is a hard problem. Since we know how to delete a BST node with none or one child, we can convert “deleting a node with two children” problem into a simpler problem by overwriting 50 with another node’s value. Which nodes can be used to overwrite 50 and still maintain the BST ordering?

d. Which node would the TreeNode’s findSuccessor method return for succ if we are deleting 50 from the BST?

2. When the findSuccessor method is called how many children does the self node have?

3. How could we improve the findSuccessor method?

4. When the spliceOut method is called from remove how many children could the self node have?

5. How could we improve the spliceOut method?
6. The shape of a BST depends on the order in which values are added (and deleted).
   a) What would be the shape of a BST if we start with an empty BST and insert the sequence of values:
      70, 90, 80, 5, 30, 110, 95, 40, 100

b) If a BST contains n nodes and we start searching at the root, what would be the worst-case big-oh \(O()\) notation for a successful search? (Draw the shape of the BST leading to the worst-case search)

7. We could store a BST in an array like we did for a binary heap, e.g. root at index 1, node at index \(i\) having left child at index \(2 \times i\), and right child at index \(2 \times i + 1\).
   a) Draw the above BST (after inserting: 70, 90, 80, 5, 30, 110, 95, 40, 100) stored in an array (leave blank unused slots)

   ![Diagram of BST](base64_encoded_diagram)

b) What would be the worst-case storage needed for a BST with \(n\) nodes?

8. a) If a BST contains \(n\) nodes, draw the shape of the BST leading to best, successful search in the worst case.

b) What is the worst-case big-oh \(O()\) notation for a successful search in this “best” shape BST?
Test 2 will be Thursday November 3 in class. It will be closed-book and notes, except for one 8.5” x 11” sheet of paper containing any notes that you want. (Yes, you can use both the front and back of this piece of paper.) Plus, you can use your Python Summary handout.

The test will cover Chapters 4 and 5. The following topics (and maybe more) will be covered:

**Chapter 4: Recursion**
- Recursive functions: base-case(s), recursive case(s), tracing recursion via run-time stack or recursion tree, “infinite recursion”
- Costs and benefits of recursion
- Recursive examples: countDown, OrderedList __str__ method, fibonacci, factorial, binomial coefficient
- Divide-and-Conquer technique of solving a problem. Examples: fibonacci, coin-change problem
- Backtracking technique of solving a problem: Examples: coin-change problem, maze (textbook)
- General concept of dynamic programming solutions for recursive problems that repeatedly solve the same smaller problems over and over. Example fibonacci, coin-change problem, binomial coefficient

**Chapter 5: Searching and Sorting**
- Sequential/Linear search: code and big-oh analysis
- Binary Search: code and big-oh analysis
- Python List implementation (ListDict) of dictionaries and big-oh analysis
- Hashing terminology: hash function, hash table, collision, load factor, chaining/closed-address/external chaining, open-address with some rehashing strategy: linear probing, quadratic probing, primary and secondary clustering
- Hashing implementation of dictionaries (ChainingDict and OpenAddrHashDict) and their big-oh analysis
- General idea of simple sorts
- Simple sorts: selection, bubble, insertion sorts and their big-oh analysis
- Advanced sorts and their big-oh analysis: heap sort, quick sort and merge sort
Question 1. (10 points) What is printed by the following program? Output:

```python
def recFn(a, b):
    if a == b:
        return 100
    elif a > b:
        return 10
    else:
        return recFn(a + 1, b - 2) + a

print("result =", recFn(1, 9))
```

Question 2. (10 points) Write a recursive Python function to compute the following mathematical function, G(n):

- G(n) = n for all value of n ≤ 1
- G(2) = 5 if n = 2
- G(n) = G(n-5) + G(n-3) + G(n-2) for all n values > 2.

```python
def G(n):
```

Question 3. a) (7 points) For the above recursive function G(n), complete the calling-tree for G(7).

```
G(7)
  G(2)
  G(4)
  G(5)
```

b) (2 point) What is the value of G(7)?

c) (1 point) What is the maximum height of the run-time stack when calculating G(7) recursively?
Question 3. The insertion sort code discussed in class is:

```python
def insertionSort(myList):
    for firstUnsortedIndex in range(1, len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1
        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1
        myList[testIndex + 1] = itemToInsert
```

Consider the following `insertMergeSort` code which calls the above `insertionSort` code twice with copies of each half of the array, and then merges the two sorted halves back together using the merge code from merge sort.

```python
def insertMergeSort(aList):
    halfSize = len(aList) // 2
    lefthalf = aList[:halfSize]
    righthalf = aList[halfSize :]
    insertionSort(lefthalf)
    insertionSort(righthalf)

    ###### BELOW IS THE MERGE CODE FROM MERGE SORT ######
    i=0  # index into lefthalf
    j=0  # index into righthalf
    k=0  # index into aList
    while i<len(lefthalf) and j<len(righthalf):  # compare and copy until one half runs out
        if lefthalf[i]<righthalf[j]:
            aList[k]=lefthalf[i]
            i=i+1
        else:
            aList[k]=righthalf[j]
            j=j+1
        k=k+1

    while i<len(lefthalf):  # copy the remaining items from lefthalf if any
        aList[k]=lefthalf[i]
        i=i+1
        k=k+1

    while j<len(righthalf):  # copy the remaining items from righthalf if any
        aList[k]=righthalf[j]
        j=j+1
        k=k+1
```

Consider the following timing of `insertionSort` vs. `insertMergeSort` on lists of 10000 elements.

<table>
<thead>
<tr>
<th>Initial arrangement of list before sorting</th>
<th>insertionSort - at the top of page</th>
<th>insertMergeSort - modified version in middle of the page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted in descending order: 10000, 9999,..., 2, 1</td>
<td>14.3 seconds</td>
<td>7.1 seconds</td>
</tr>
<tr>
<td>Already in ascending order: 1, 2,..., 9999, 10000</td>
<td>0.005 seconds</td>
<td>0.009 seconds</td>
</tr>
<tr>
<td>Randomly ordered list of 10000 numbers</td>
<td>7.4 seconds</td>
<td>3.6 seconds</td>
</tr>
</tbody>
</table>

a) (10 points) Explain why `insertMergeSort` (modified version in middle of page) out performs the original `insertionSort`.

b) (10 points) In either version, why does sorting the randomly order list take about halve the time of sorting the initially descending ordered list?
Question 4. (20 points) In insertion sort the inner-loop takes the "first unsorted item" (25 at index 6 in the below example) and "inserts" it into the sorted part of the list "at the correct spot."

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unssorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>10 20 35 40 45 60 25 50 90 • • •</td>
</tr>
</tbody>
</table>

In class we discussed the following insertion sort code which sorts in ascending order (smallest to largest) and builds the sorted part on the left-hand side of the list, i.e.:

```python
def insertionSort(myList):
    for firstUnsortedIndex in range(1, len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1
        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1
        myList[testIndex + 1] = itemToInsert
```

For this question write a variation of the above insertion sort that:
- sorts in **descending order** (largest to smallest)
- builds the sorted part on the right-hand side of the list, i.e.,

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>• • 25 40 90 60 50 45 35 20 10</td>
</tr>
</tbody>
</table>

```python
def insertionSortVariation(myList):
```
Question 5. Two common rehashing strategies for open-address hashing are linear probing and quadratic probing:

<table>
<thead>
<tr>
<th>quadratic probing</th>
<th>Check the square of the attempt-number away for an available slot, i.e.,</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[\text{home address} + (\text{rehash attempt} #)^2 + (\text{rehash attempt} #) / 2] % (hash table size), where the hash table size is</td>
</tr>
<tr>
<td></td>
<td>a power of 2. Integer division is used above</td>
</tr>
</tbody>
</table>

a) (8 points) Insert “Paul Gray” and then “Sarah Diesburg” using Linear (on left) and Quadratic (on right) probing.

```
Hash Table with Linear Probing
0  Ben Schafer
1  
2  
3  Philip East
4  
5  
6  Mark Fienup
7  John Doe

Hash function
hash(John Doe) = 7
hash(Philip East) = 3
hash(Mark Fienup) = 6
hash(Ben Schafer) = 0
hash(Paul Gray) = 6
hash(Sarah Diesburg) = 7
```

b) (7 points) Explain why both linear and quadratic probing both suffer from primary clustering?

Question 6. Recall the general idea of Heap sort which uses a min-heap (class BinHeap with methods: BinHeap(), insert(item), delMin(), isEmpty(), size()) to sort a list.

**General idea of Heap sort:**

1. Create an empty heap
2. Insert all n list items into heap
3. delMin heap items back to list in sorted order

```
myList  unsorted list with n items

heap with n items

myList  sorted list with n items
```

a) (10 points) Complete the code for heapSort so that it sorts in descending order

```
from bin_heap import BinHeap
def heapSort(myList):
    myHeap = BinHeap()  # Create an empty heap
```

b) (5 points) Determine the overall \(O()\) for your heap sort and briefly justify your answer.