# Data Structures

Topic #7

## Today’s Agenda

- **How to measure the efficiency of algorithms?**
- **Discuss program #3 in detail**
- **Review for the midterm**
  - what material to study
  - types of questions
  - walk through a sample midterm

## Algorithm Efficiency

- So far, we have discussed the efficiency of various data structures and algorithms in a subjective manner
- Instead, we can measure the efficiency using mathematical formulations using the Big O notation
- This allows us to more easily compare and contrast the run-time efficiency between algorithms and data structures

## Algorithm Efficiency

- If we say: Algorithm A requires a certain amount of time proportional to f(N)...
  - this means that regardless of the implementation or computer, there is some amount of time that A requires to solve the problem of size N.
  - Algorithm A is said to be order f(N) which is denoted as O(f(N));

## Algorithm Efficiency

- f(N) is called the algorithm's growth-rate function.
- We call this the BIG O Notation!
- Examples of the Big O Notation:
  - If a problem requires a constant time that is independent of the problem's size N, then the time requirement is defined as: O(1).

## Algorithm Efficiency

- If a problem of size N requires time that is directly proportional to N,
  - then the problem is O(N).
- If the time requirement is directly proportion to Nsquared,
  - then the problem is O(Nsquared), etc.

# Algorithm Efficiency

- Whenever you are analyzing these algorithms,
  - it is important to keep in mind that we are only interested in significant differences in efficiency.
  - Can anyone tell me if there are significant differences between an unsorted array, linear linked list, or hash table implementation of retrieve (for a “table” abstraction)??

## Algorithm Efficiency

- Notice that as the size of the list grows,
  - the unsorted array and pointer base implementation might require more time to retrieve the desired node (it definitely would in the worst case situation...because the node is farther away from the beginning of the list).
  - In contrast, regardless of how large the list is, the hash table implementation will always require the same constant amount of time.

## Algorithm Efficiency

- Therefore, the difference in efficiency is worth considering if your problem is large enough.
- However, if your list never has more than a few items in it, the difference is not significant!

## Algorithm Efficiency

- There is one side note that we should consider.
  - When evaluating an algorithm's efficiency, we always need to keep in mind the trade-offs between execution time and memory requirements.
  - The Big O notation is denoting execution time and does not fill us in concerning memory requirements and/or algorithm limitations.

## Algorithm Efficiency

- So, evaluate your performance needs and...
  - consider how much memory one approach requires over another
  - evaluate the strengths/weaknesses of the algorithms themselves (are there certain cases that are not handled effectively?).
  - Overall, it is important to examine algorithms for both style and efficiency. If your problem size is small, don't over analyze; pick the algorithm easiest to code and understand. Sometimes less efficient algorithms are more appropriate.

# Algorithm Efficiency

- Some things to keep in mind when using this notation:
  - You can ignore low-order terms in an algorithm's growth rate.
  - For example, if an algorithm is $O(N^3+ 4*N^2+3*N)$ then it is also O(N3). Why?
  - Because $N^3$ is significantly lager than either $4*N^2$ or $3*N$...especially when N is large.
  - For large N values...the growth rate of $N^3+ 4*N^2+3*N$ is the same as $N^3$

## Algorithm Efficiency

- Also, you can ignore a constant being multiplied to a high-order term.
- For example: if an algorithm is $O(5*N^3)$, then it is the same as $O(N^3)$.
- However, not all experts agree with this approach
  - and there may be situations where the constants have significance

## Algorithm Efficiency

- Lastly, one algorithm might require different times to solve different problems that are of the same size.
  - For example, searching for an item that appears in the first location of a list will be finished sooner than searching for an item that appears in the last location of the list (or doesn't appear at all!).

## Algorithm Efficiency

- Therefore, when analyzing algorithms,
  - we should consider the maximum amount of time that an algorithm can require to solve a problem of size N -- this is called the **worst case**.
  - Worst case analysis concludes that your algorithm is O(f(N)) in the worst case.

## Algorithm Efficiency

- You might also consider looking at your algorithm time requirements using **average case** analysis.
  - This attempts to determine the average amount of time that an algorithm requires to solve problems of size N.
  - In general, this is far more difficult to figure out than worst case analysis.

## Algorithm Efficiency

- This is because you have to figure out the probability of encountering various problems of a certain size and the distribution of the type of operations performed.
- Worst case analysis is far more practical to calculate and therefore it is more common.

## Algorithm Efficiency

- The next step is to learn how to figure out an algorithm's growth rate.
- We know how to denote it...and we know what it means (i.e., usually the worst case) and we know how to simplify it (by not including low order terms or constants)
  - ...but how do we create it?

## Algorithm Efficiency

- Here is an example of how to analyze the efficiency of an algorithm to traverse a linked list...

```
void printlist(node *head)
{
    node * cur;
    cur = head;
    while (cur != NULL) {
        cout <<cur->data;
        cur = cur->link;     }
```

## Algorithm Efficiency

- If there are N nodes in the list;
  - the number of operations that the function requires is proportional to N.
  - For example, there are N+1 assignments and N print operations, which together are 2*N+1 operations.
  - According to the rules we just learned about, we can ignore both the coefficient 2 and the constant 1; they are meaningless for large values of N.

## Algorithm Efficiency

- Therefore, this algorithm's efficiency can be denoted as O(N);
  - the time that printlist requires to print N nodes is proportional to N.
  - his makes sense: it takes longer to print or traverse a list of 100 items than it does a list of 10 items.

# Algorithm Efficiency

- Another example, using a nested loop:

```
for (i=1; i <= n; i++)
     for (j=1; j <=n; j++)
          x = i*j;
```

- This is O(n squared)

## Algorithm Efficiency

- The concepts learned here can also be used to help choose the type of ADT to use and how efficient it will be.
- For example, when considering whether to use arrays or linked lists, you can use this type of analysis
  - ...since there may be significant difference in the efficiency between the two!

## Algorithm Efficiency

- Take, for example, the ADTs for the ordered list operation RETRIEVE;
  - remember, it retrieves a value of the item in the Nth position in the ordered list.
  - In the array based implementation, the Nth item can be accessed directly (it is stored in position N). This access is INDEPENDENT OF N!
  - Therefore, RETRIEVE takes the same amount of time to access either the 100th item or the first item in the list. Thus, an array based implementation of RETRIEVE is O(1).

# CALCULATE BEST/WORST

- So, let’s evaluate what the Big O would be
  - for an absolute ordered list using an array

    retrieve: remove: insert:
  - for a relative ordered list using an array

    retrieve: remove: insert:
  - for an absolute ordered list using a LLL

    retrieve: remove: insert:
  - for an relative ordered list using a LLL

    retrieve: remove: insert:

# Continue....BEST/WORST

- So, let's evaluate what the Big O would be
  - for a table ADT using an unsorted array:

    retrieve:  remove:  insert:
  - for a table ADT using an unsorted LLL:

    retrieve:  remove:  insert:
  - for a table ADT using a sorted array:

    retrieve:  remove:  insert:
  - for a table ADT using a hash table:

    retrieve:  remove:  insert:

## Discuss Program #3

- **Program #3**
  - expects that you are able to build two different hash tables for a “table ADT”
  - needs to isolate the client program from knowing that hashing is being performed
  - where the class needs two data members for two different hash tables
  - and two different data members for the sizes of these hash tables

# Discuss Program #3

- **Program #3**
  - why might the sizes of the hash tables be different when the number of items in each table will be the same???
  - remember the hash tables, since we are implementing chaining need to be arrays of pointers to nodes
  - remember that the constructor needs to allocate these arrays and then initialize each element of the arrays to null

## Discuss Program #3

- **Program #3**
  - what is most important is developing a technique for FAST retrieval by either key value
  - which is why two hash tables are being used
  - but...at the same time we don’t want to duplicate our data so make sure that the data only occurs once and that each node <u>points</u> to the data desired

## Discuss Program #3

- **Program #3**
  - Remember that you destructor needs to deallocate the data (only deallocate the data once...more than once may lead to a segmentation fault!)
  - deallocate the nodes for both hash tables
    - (the nodes for one hash table will be DIFFERENT than the nodes for the 2nd hash table)
  - and, deallocate the two hash tables

## Discuss Midterm

- **Review for the Midterm**
  - The midterm is closed book, closed notes
  - it will cover position oriented abstractions such as stacks, queues, absolute ordered lists and relative ordered lists
  - it will cover array, linear linked list, circular linked list, and doubly linked list representations
  - it may also cover dummy head node and derivations of the standard linked list

## Discuss Midterm

- **To prepare...**
  - I recommend walking through the self check exercises in the book for the chapters that have been assigned
  - Answer the self-check exercises and compare your results with other members in the class
  - Practice writing code. Re-do your answers for homework #1...very important!

## Discuss Midterm

- **For example...**
  - Can you make a copy of a linear linked list?
  - What about deallocating all nodes in a circular linked list
  - Can you find the largest data item that resides within a sorted linked list? How about an unsorted linked list?
  - Could you do the same thing with an array of linked lists (unsorted, of course)

# Discuss Midterm

- **Or...**
  - Can you determine if two linked lists are the same?
  - How about copying the data from a linked list into an array...or vice versa?
  - Can you determine if the data in an array is the same as the data in a linked list?
  - Would your answer change if you were comparing a circular array to a circular linked list?