CSCI 1900
Discrete Structures

Complexity
Reading: Kolman,
Sections 4.6, 5.2, and 5.3

Comparing Algorithms

This presentation uses an example to present the material from the following three sections of our text book.

- Section 4.6 – Computer Representations
- Section 5.2 – Hashing
- Section 5.3 – Comparing algorithms based on computational method

Example – Access Control System

This entire lecture is based on the example of a database listing valid student id's to be used in an access control system. If the id is in the database, then the student is allowed in the door. Each method of storing the data is compared based on the complexity it takes to store a new record and what it takes to search for a record.

Unsorted lists – new record

Each time a new student id is added, simply add it to the end of the list:

```
students[next_empty_entry] = NEW_ID
next_empty_entry = next_empty_entry + 1
```


Unsorted lists – search records

To search for an id, simply go through the list looking for a match

```
found = 0
for count=0 to (next_empty_entry-1)
  if students[count] = FIND_ID
     found = 1
     exit for
  endif
next count
```


Cycle Counts

In an effort to estimate the relative speeds of each of these situations, assume the following cycle counts for each instruction.

- assignment/equals = 3 cycles
- increment = 1 cycle
- for loop or while loop = 3 cycles
- if = 2 cycles
- switch/case = 4 cycles

Unsorted lists
Cycles for new record

To add an id, there is one assignment and one increment for a total of 4 cycles

Unsorted lists
Cycles for searching records

To search, assume that on average, we have to go through half of the items before finding a match

- 1 assignment = 3 cycles
- N/2 loops = N/2 * 3 cycles
- N/2 ifs = N/2 * 2 cycles
- 1 assignment in if = 3 cycles
- Total = (5 * N/2) + 6

Sorted list

The best sorted list is one where there is a unique, sequential key that identifies the argument to the array.

Sorted array lists – new record

Depending on the algorithm used to derive the index, a sorted list that has ascending indexes could be a bit more complex.

```
index=index from NEW_ID
for i=next_empty_entry to (index+1) step -1
students[i] = students[i-1]
next i
students[index] = NEW_ID
next_empty_entry = next_empty_entry + 1
```


Sorted array lists – new record

...or it could be a great deal simpler by making the size of the array equal to the size of the set of all possible keys.

```
index=index from NEW_ID
students[index] = NEW_ID
```

For us this isn't practical, i.e., our array size would need to be able to contain 10^9 records.

Sorted array lists – search records

For a sorted list that has ascending indexes, the search algorithm is simple.

```
found = 0
index=index from FIND_ID
if students[index] = FIND_ID
  found = 1
end if
```

Sorted array lists
Cycles for adding record

To add an id, there is an assignment followed by a for loop with, on average, N/2 loops. Inside the for loop, there is an assignment, and since the for loop is executed an average of N/2 times, there are N/2 assignments. The for loop is followed by an assignment and an increment. This gives us a total of:

3 + (N/2*3) + (N/2*3) + 3 + 1 = 3*N + 7

Sorted array lists
Cycles for searching records

To find a record, there are two assignments followed by an if and then a possible third assignment. This gives us a total of:

3 + 3 + 2 + 3 = 11 cycles

Sorted list – using pointers

We don't have a sorted index here. Our unique key could take on any value from 000-00-0000 to 999-99-9999.

Instead, what we will do is make 10 lists, each representing a student id that begins with a different decimal digit.

Sorted lists – new record

For the case where we have created ten lists, we've basically divided the unsorted list into ten. Each time a new student id is added, find the list it should be in, and add it to the end of the list

```
index = first digit of NEW_ID
students[index, next_empty_entry[index]] =
  NEW_ID
next_empty_entry[index] = next_empty_entry
  [index] + 1
```


Sorted lists – search records

This is the same as finding a record in an unsorted list except that we need to select which list to search from.

```
index = first digit of FIND_ID
found = 0
for count=0 to (next_empty_entry[index]-1)
  if students[index, next_empty_entry[index]] =
  FIND_ID
      found = 1
  exit for
  endif
next count
```


Sorted lists
Cycles for adding record

To add an id, there are two assignments, index and NEW_ID, and one increment for a total of 7 cycles

Sorted lists
Cycles for search records

- This is exactly the same as for the unsorted lists except for two things:
  - We need to assign the value of index (add 3 cycles)
  - The lists are about one tenth the length of the full list
- This gives us a new total of :
  1/10 * (5 * N/2) + 9

Unbalanced lists

At ETSU, however, the list for student id's beginning with '4' would overwhelm the other lists, and the benefit would be negligible.

Hashing

In order to determine which list a student id should be assigned to, we need to create a function that maps each student id to a different list in a balanced way. The function needs to be reproducible so that when we go to find the record, it is in the list we originally assumed it would be in.

Hashing

A hashing algorithm randomly assigns an integer to one of *n* lists. The typical hashing algorithm is a "mod-*n*" function, i.e., a function that determines the remainder of an integer after a division by *n*. By using a mod-*n* function, we can have as many lists as we want.

Cycle count for hashing

The cycle count for hashing should be very close to the cycle count for the sorted list example with two difference:

- Assigning the value for index will take more cycles because the mod-*n* involves division
- The number of lists is up to our selection of *n*. Therefore, we could significantly shorten up the search time if we have the memory to add more lists.

Linked Lists

There is a problem with the sorted and hashed lists presented earlier. A certain amount of memory must be allocated to each list. For most of the lists, this is wasted space. For some of the lists, this may be a limiting factor forcing some student id's to be refused because of no space while other lists have plenty of space.

Linked Lists (continued)

A linked list adds a second array to tell the computer where to find the next record in the sorted list. This way, the next element does not have to be stored sequentially in memory.

Linked Lists (continued)

- Corresponding index in second array indicates index of next element in list
- If index of next element is listed as zero, that is the end of the list
- This means that the index count must start at 1
- A separate variable must point to beginning of list
- Sometimes a third array is added so that we can go backwards in the list, i.e., corresponding index in third array points to index of previous element.

In-class exercise

- See Figures 4.30 and 4.33 from the textbook
- Write the BASIC code to do the following:
  - Add a record to a linked list
  - Find a record in a linked list
- Estimate the number of cycles required for both adding and finding a record.