Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides adapted from slides of Textbook, Dr. Ramirez's CS 0445, and Dr. Farnan's CS 1501)

Announcements

- Upcoming Deadlines
  - Homework 11 and Lab 11: next Monday 12/12
  - Assignment 3: Friday 12/16 @ 11:59 pm
  - Assignment 4: Friday 12/16 @ 11:59 pm
  - Lab 12 and Homework 12: Monday 12/19

Bonus Opportunities

- Bonus Lab (1%) and homework (2%) due on 12/19
- Assignment 5 is bonus (4%) and is due on 12/19
- 1 bonus point for entire class when OMETs response rate >= 80%
  - Currently at 23%
  - Deadline is Sunday 12/11

Final Exam

- Same format as midterm
- Non-cumulative
- Date, time and location on PeopleSoft
  - Thursday 12/15 8-9:50 am (coffee served!)
- Same classroom as lectures
- Study guide and practice test to be posted soon

Previous Lecture …

- Hashing!
  - Handling collisions
    - Open addressing
      - Double hashing
    - Closed addressing
- Code walkthrough of Hash Table implementation
- String matching
  - brute-force algorithm

This Lecture …

- String matching
  - Brute-force
  - Boyer Moore
  - Rabin Karp
- ADT Queue

Muddiest Points

- **Q: why do we have iterable interface and iterator interface. As only iterator works here**
- Iterator interface is used to implement iterators
- Iterable interface is used to implement containers that have iterators
  - allows us to use the for-each loop structure

```
IterableLinkedList<Integer> list = new …..
for(Integer x : list){
   //do something with x
}
```

Muddiest Points

- **Q: Can we please get more in class tophat questions? It would be a very helpful way to boost our grades.**
- Sure. Let's have a couple today and next lecture!

String Matching

- Have a pattern string p of length m
- Have a text string t of length n
- Can we find an index i of string t such that each of the m characters in the substring of t starting at i matches each character in p
  - Example: can we find the pattern "fox" in the text "the quick brown fox jumps over the lazy dog"?
    - Yes! At index 16 of the text string!

Simple approach

- BRUTE FORCE
  - Start at the beginning of both pattern and text
  - Compare characters left to right
  - Mismatch?
  - Start again at the 2nd character of the text and the beginning of the pattern…

Brute force code

```
public static int bf_search(String pat, String txt) {
    int m = pat.length();
    int n = txt.length();
    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (txt.charAt(i + j) != pat.charAt(j))
                break;
        }
        if (j == m)
            return i; // found at offset i
    }
    return n; // not found
}
```

Alternate implementation of Brute-Force Algorithm

```
public static int bf_search(String pat, String txt)
{
   int j, m = pat.length();
   int i, n = txt.length();
   for (i = 0, j = 0; i <= n – m && j < m; i++) {
     if (txt.charAt(i) == pat.charAt(j))
          j++;
     else { i -= j; j = 0; }
   }
   if (j == m)
        return i - m; // found at offset i
   else return n; // not found
}
```

Tracing Brute force Algorithm

i:	0							
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:	0							

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n – m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:	0	1						
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:	0	1						

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:		1	2					
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:		1	2					

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:			2	3				
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:			2	3				

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:				3	4			
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:				3	4			

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:					4	5		
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:					4	5		

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:	0					5		
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:	0					5		

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
        if (txt.charAt(i) == pat.charAt(j))
            j++;
        else { i -= j; j = 0; }
    }
    if (j == m)
        return i - m; // found at offset i
    else return n; // not found
}
```

i:	0	1						
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:	0							

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:		1						
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:	0							

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:		1	2					
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:	0							

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:			2	3				
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:	0	1						

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:				3	4			
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:		1	2					

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:					4	5		
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:			2	3				

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:						5	6	
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:				3	4			

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i <= n - m && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:							6	7
text:	A	B	A	B	A	B	A	C
pattern:	A	B	A	B	A	C		
j:					4	5		

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i < n && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:								7	8
text:	A	B	A	B	A	B	A	C	
pattern:	A	B	A	B	A	C			
j:						5	6		

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i < n && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

i:			↓						8
text:	A	B	A	B	A	B	A	C	
pattern:	A	B	A	B	A	C			
j:							6		

```
public static int bf_search(String pat, String txt)
{
    int j, m = pat.length();
    int i, n = txt.length();
    for (i = 0, j = 0; i < n && j < m; i++) {
       if (txt.charAt(i) == pat.charAt(j))
              j++;
       else { i -= j; j = 0; }
    }
    if (j == m)
           return i - m; // found at offset i
    else return n; // not found
}
```

Brute force analysis

- Runtime?
  - What does the worst case look like?
    - t = XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXY
    - p = XXXXY
  - m (n - m + 1)
    - O(nm) if n >> m
  - Is the average case runtime any better?
    - Assume we mostly mismatch on the first pattern character
    - O(n + m)
      - Θ(n) if n >> m

Where do we improve?

- Improve worst case
  - Theoretically very interesting
  - Practically doesn't come up that often for human language
- Improve average case
  - Much more practically helpful
    - Especially if we anticipate searching through large files

Improve Average Case: Boyer Moore

- What if we compare starting at the end of the pattern?
  - `t = ABCDVABCDWABCDXABCDYABCDZ`
  - `p = ABCDE`
  - V does not match E
    - Further V is nowhere in the pattern…
    - So skip ahead m positions with 1 comparison!
      - Runtime?
        - In the best case, n/m
- When searching through text with a large alphabet, will often come across characters not in the pattern.
  - One of Boyer Moore's heuristics takes advantage of this fact
    - Mismatched character heuristic

Mismatched character heuristic

- How well it works depends on the pattern and text at hand
  - What do we do in the general case after a mismatch?
    - Consider:
      - `t = XYXYXYZXXXXXXXXXXXXXX`
      - `p = XYXYZ`
    - If mismatched character *does* appear in p, need to "slide" to the right to the next occurrence of that character in p
      - Requires us to pre-process the pattern
        - Create a right array

Pattern: A B C D E

right = [0, 1, 2, 3, 4, -1, -1, …]

```
for (int i = 0; i < R; i++)
   right[i] = -1;
for (int j = 0; j < m; j++)
   right[p.charAt(j)] = j;
```

Mismatched character Procedure

- Let j be the index in the pattern currently under comparison
- At mismatch, slide pattern to the right by
  - j - right[mismatched_text_char] positions
  - If < 1, slide 1

Mismatched character heuristic example

Text: A B C D X A B C D C A B C D Y A E C D E A B C D E

A B C D E

A B C D E

A B C D E

A B C D E

A B C D E

A B C D E

A B C D E

Pattern: A B C D E

right = [0, 1, 2, 3, 4, -1, -1, …]

Runtime for mismatched character

- What does the worst case look like?
  - Runtime:
    - $\Theta(nm)$
      - Same as brute force!
- This is why mismatched character is only one of Boyer Moore's heuristics
  - Another works similarly to KMP
- See BoyerMoore.java

Let’s use hashing!

- Hashing was cool, let's try using that

```
public static int hash_search(String pat, String txt) {
    int m = pat.length();
    int n = txt.length();
    int pat_hash = h(pat);
    for (int i = 0; i <= n - m; i++) {
            if (h(txt.substring(i, i + m)) == pat_hash)
                return i; // found!
    }
    return n; // not found
}
```

Well that was simple

- Is it efficient?
  - Nope! Practically worse than brute force
    - Instead of nm character comparisons, we perform n hashes of m character strings
- Can we make an efficient pattern matching algorithm based on hashing?

Horner's method

- Brought up during the hashing lecture

```
public long horners_hash(String key, int m) {
      long h = 0;
      for (int j = 0; j < m; j++)
            h = (R * h + key.charAt(j)) % Q;
      return h;
}
```

Can we compute the hash of the next m characters using the hash of the previous m characters in O(1) time?

Efficient hash-based pattern matching

```
text = "abcdefg"
pattern = "defg"
```

- This is Rabin-Karp

What about collisions?

- Note that we're not storing any values in a hash table…
  - So increasing Q doesn't affect memory utilization!
    - Make Q really big and the chance of a collision becomes really small!
      - But not 0…
- OK, so do a character by character comparison on a hash match just to be sure
  - Worst case runtime?
    - Back to brute force esque runtime…

Assorted casinos

- Two options:
  - Do a character by character comparison after hash match
    - Guaranteed correct
    - Probably fast

Las Vegas

  - Assume a hash match means a substring match
    - Guaranteed fast
    - Probably correct

Monte Carlo

ADT Queue

- Queue
  - Data is **added to the end** and **removed from the front**
  - Logically the items other than the front item cannot be accessed
    - Think of a bowling ball return lane
      - Balls are put in at the end and removed from the front, and you can only see / remove the front ball
  - Fundamental Operations
    - enqueue an item to the end of the queue
    - dequeue an item from the front of the queue
    - front – look at the top item without disturbing it

Queues

- A Queue organizes data by First In First Out, or FIFO (or LILO – Last In Last Out)
- Like a Stack, a Queue is a simple but powerful data structure
  - Used extensively for simulations
    - Many real life situations are organized in FIFO, and Queues can be used to simulate these
    - Allows problems to be modeled and analyzed on the computer, saving time and money

Uses of Queues: Simulation

- Ex: A bank wants to determine how best to set up its lines to the tellers:
  - Option 1: Have a separate line for each teller
  - Option 2: Have a single line, with the customer at the front going to the next available teller
  - How can we determine which will have better results?
    - We can try each one for a while and measure
      - Obviously this will take time and may create some upset customers
    - We can simulate each one using reasonable data and compare the results
- Other (often more complex) problems can also be solved through simulation
- Check Lab 12 code!

Implementing Queues

- Queue Implementation?
  - We need a structure that has access to both the front and the rear
  - We'd like both enqueue and dequeue to be O(1) operations
  - We have two basic approaches:
    - Use a linked-list based implementation
    - Use an array based implementation
  - Let's consider each one

Linked Queues

- Queue using a Linked List
  - This implementation is fairly straightforward as long as we have a **doubly linked list** or access to the front and rear of the list
    - enqueue simply adds a new object to the end of the list
    - dequeue simply removes an object from the front of the list
    - Other operations are also simple
    - We can build our Queue from a LinkedList object, making the implementation even simpler
      - This is more or less done in the JDK

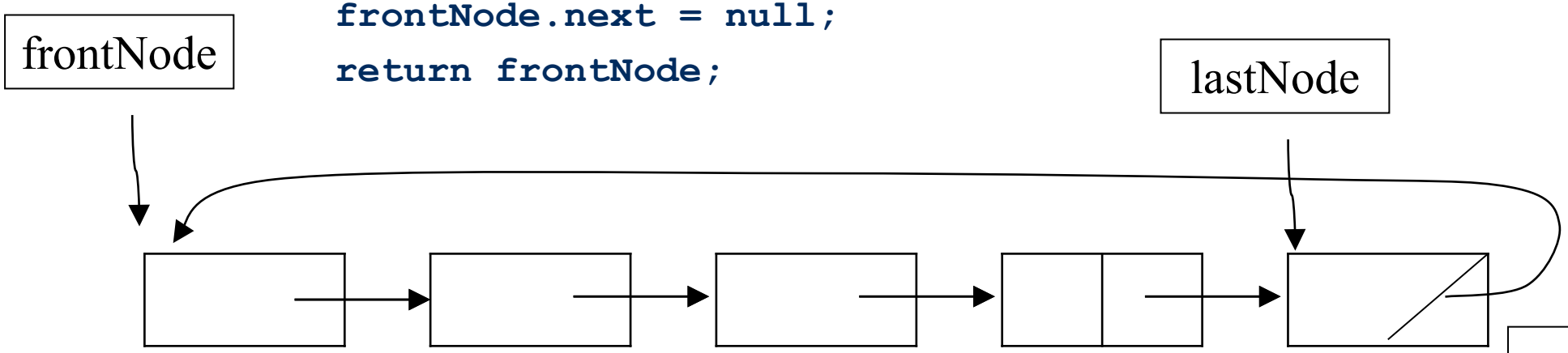
Linked Queues

- Are there other linked options?
  - a circular linked list
    - The extra link gives us all the functionality we need for a Queue
      - **enqueue?**

```
newNode = new Node(newEntry, lastNode.next);
lastNode.next = newNode;
lastNode = newNode;
```

      - **dequeue?**

```
frontNode = lastNode.next;
lastNode.next = frontNode.next;
frontNode.next = null;
return frontNode;
```



Array Queues

- Queue using an array
  - Arrays that we have seen so far can easily add at the end, so enqueue is not a problem
    - Can clearly be done in O(1) time
    - We may have to resize, but we know how to do that too
  - However, removing from the front is trickier
    - In ArrayList, removing from the front causes the remaining objects to be shifted forward
      - This gives a run-time of O(N), not O(1) as we want
    - So we will not use an ArrayList
      - Instead we will work directly with an array to implement our Queue

Array Queues

- How can we make dequeue an O(1) operation?
  - What if the front of the Queue could "move" – not necessarily be at index 0?
    - We would then keep a head index to tell us where the front is (and a tail index to tell where the end is)
  - Ok…so now we can enqueue at the rear by incrementing the tail index and putting the new object in that location and we can dequeue in the front by simply returning the head value and incrementing the head index

Array Queues

- This implementation will definitely work, but it has an important drawback:
  - Both enqueue and dequeue increment index values
  - Once we increment front past a location, we never use that location again
  - Thus, as the queue is used the data migrates toward the end of the array
  - Clearly this is wasteful in terms of memory
- What can we do to fix this problem?
  - We need a way to reclaim the locations at the front of the array without spending too much time
    - So shifting is not a good idea
    - Any ideas?