CRICOS PROVIDER 00123M

School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Complexity

adelaide.edu.au

seek LIGHT

Overview

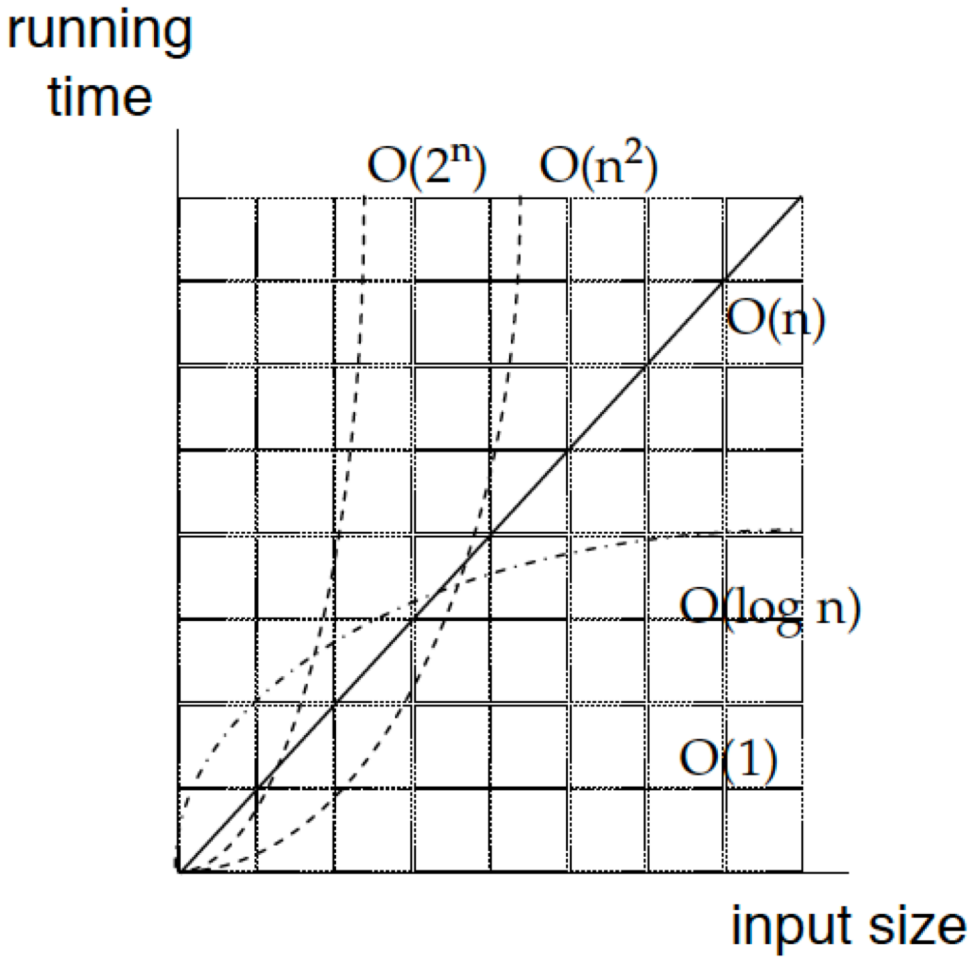
- Important growth rates
- Simple instructions about complexity
- Linked list

Review on Big O [O(g(n))]

- Assume f(n) represents the *actual* execution time of an algorithm. (do you remember my search algorithm?)
- f(n) = O(g(n)) if there exist positive constants c and n_0 such that f(n) <= c.g(n) when n>= n_0.
- 2n+2= O(n)?
- n^2= O(n)? No!
- log n= O(n)?

Typical Big-Oh Running Times

Big-Oh	Informal name
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	*n log n*
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential



How are you finding the complexities?

- We will have simple rules
  - Simple statements (Math operators: +, -, &&, *, etc …, Assignment , Array indexing, Comparisons) are all O(1)
  - The running time of a loop is at most the running time of the statements inside the loop (including tests) multiplied by the number of iterations
    - Nested loops?
  - The running time of an if/else statement is at most the running time of the test plus the larger of the running times of the statements in the if and else block.
- Recursive algorithms? Next session!

General Rules

- Some mathematical background is required for analyzing computational complexity

Rule 1. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

1. $f_1(n) + f_2(n) =$
   $O(g_1(n)+g_2(n)) = O(\max(g_1(n),g_2(n)))$
2. $f_1(n) * f_2(n) =$
   $O(g_1(n)*g_2(n))$

Rule 2. If f(n) is a polynomial of degree k, then

$$f(n) = O(n^k)$$

Rule 3. if $f(n)=\log^k n$ then $f(n)= O(n)$ for any constant k.

Simple Statement

- Simple statements:
  - Math operators: +, -, &&, *, etc …
  - Assignment , array indexing, …
  - Comparison

- The simple statements are all O(1)

- What about blocks of simple statements?

Simple Statement

- Consider the code block below

```
int next, n1, n2;

next = n1 + n2;
n2 = n1;
n1 = next;
```

- These statements are all O(1).
- They take a constant amount of time to execute, ***independent of the input size!***
- The complexity of the entire code segment is O(1).
  - These statements altogether still take a constant amount of time to execute.

Loops

- For-loops: (n is the input)

```
int counter = 0;
for(int i = 0; i< n ; i++){
  counter += i;
 }
```

- The running time of a for loop is at most the running time of the statements inside the for loop (including tests) multiplies the number of iterations.
- O(n*[complexity of statements inside the loop])
  - O(n)

Loops

```
int counter = 0;

for(int i = 0; i< 100; i++){

  counter += i;

 }
```

- The statements are performed 100 times.
- The complexity of the entire code segment is 100 * [the complexity of the statements inside the loop]
  - 100*c =O(1)

Loops

- Nested loops:

```
int counter = 0;

for(int i = 0; i< n ; i++){
  for(int j = 0; j< n ; j++){
    counter ++;
  }
 }
```

- The total running time of the statements that form na group of nested loops is the running time of the inner statements multiplied by the product of the sizes of all the loops.
- O(n^2)

Loops

- Nested loops

```
for(int i = 0; i< n; i++){
  for(int j = 0; j< m; j++){
    counter ++;
  }
 }
```

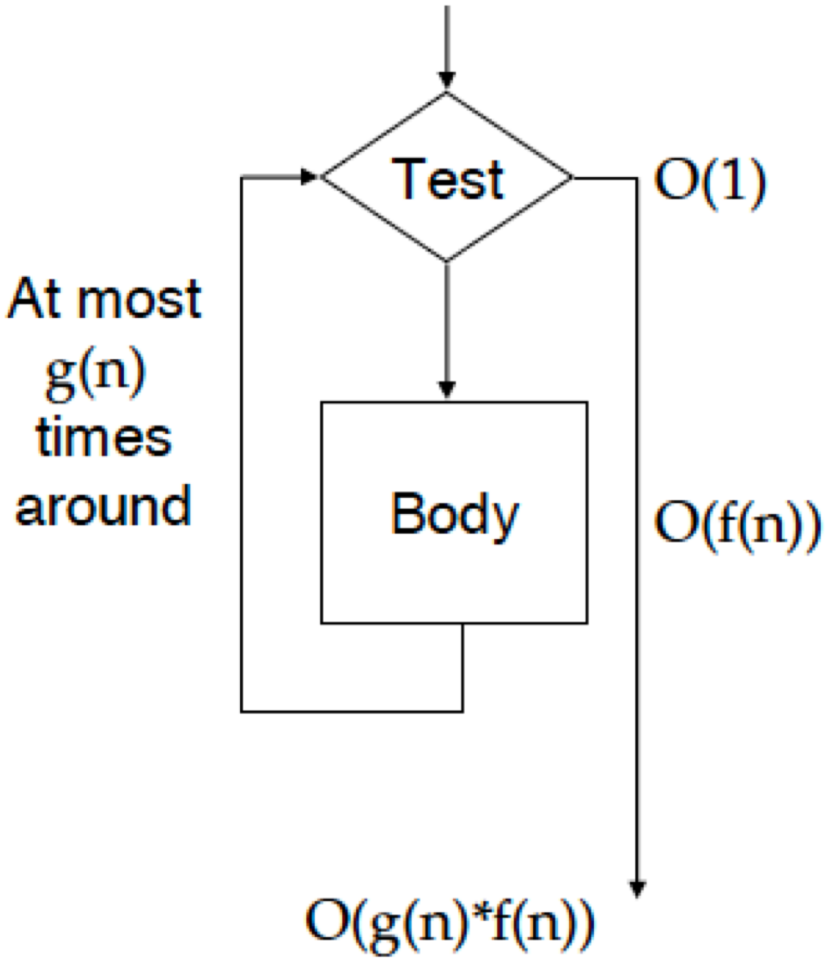
O(mn)

```
for(int i = 0; i< n; i++){
  for(int j = 0; j< 100; j++){
    counter ++;
  }
 }
```

O(n)

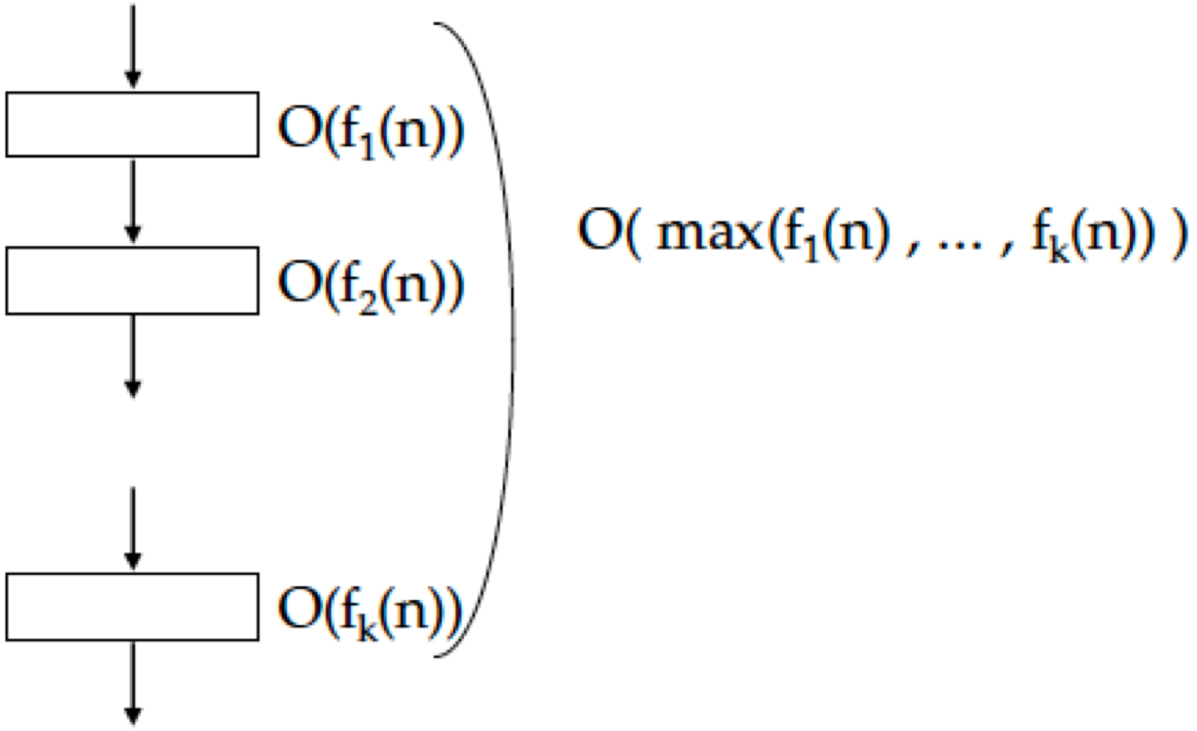
Loops

- While-loops:

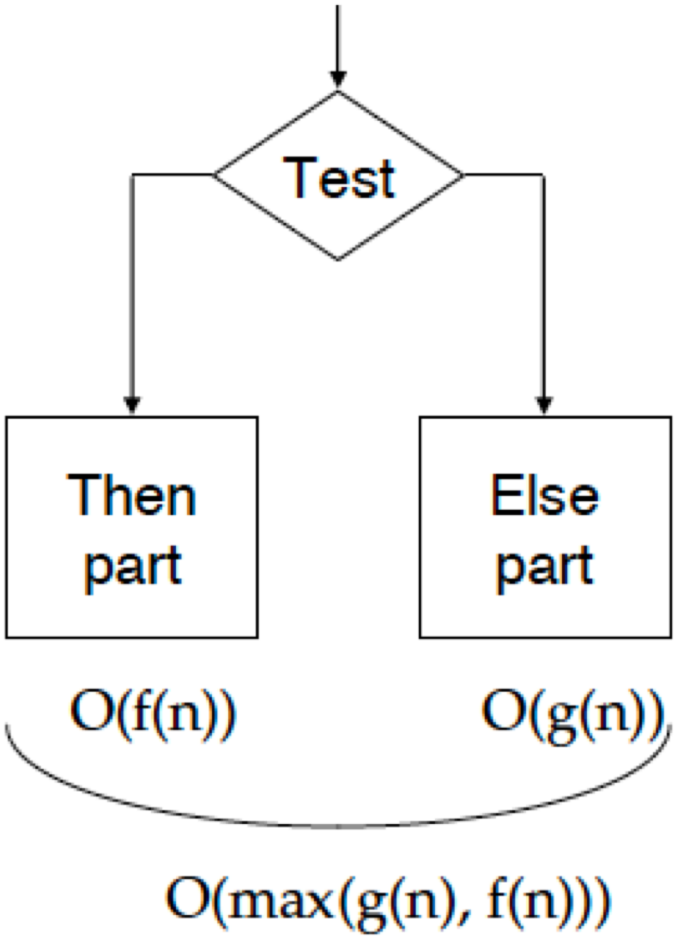


Consecutive Statements

- Block of statements without function calls is just summation.



If/else Statement



- The running time of an if/else statement is never more than the running time of the test plus the larger of the running times of the statements in the if and else block.

If/else Statement

```
if(a>b){
  for(int i=0; i<n; i++){
    counter ++;
  }
 }else{
  counter = 0;
 }
```

O(n)

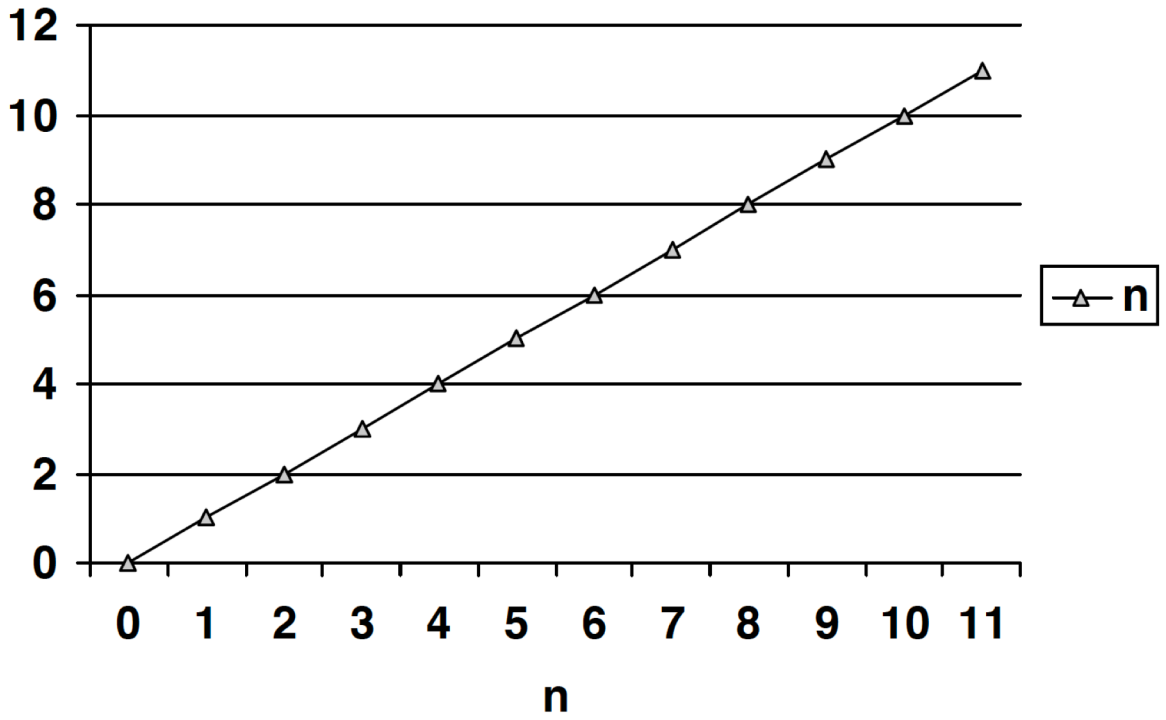
Example

- Iterative version of Fibonacci number calculation
- The program structure tree
- O(n)

```
1  int Fib(int n){
2    if(n<0){
3      return -1;
4    }
5
6    if(n == 0){
7      return 0;
8    }
9
10   if(n == 1){
11     return 1;
12   }
13
14   int n1 = 0;
15   int n2 = 1;
16   int next = 2;
17
18   for(int current = 2; current <= n; current ++){
19     next = n1 + n2;
20     n2 = n1;
21     n1 = next;
22   }
23
24   return next;
25 }
```

Example

- The iterative version of Fibonacci has a linear growth rate.
- The run time grows in proportion to the magnitude of the Fibonacci number we are computing.

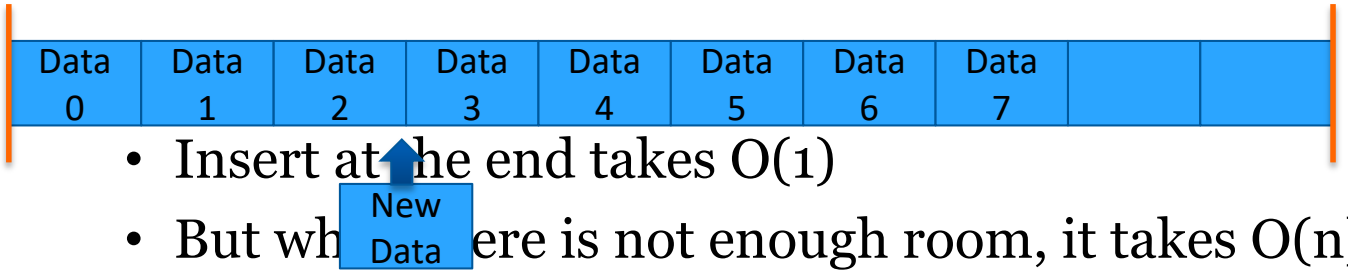


Summary

- Notations:
  - Big O: for presenting an upper bound
- Simple Rules
  - Summation and multiplication
  - Polynomials with degree k: O(n^k)
  - Analysis of Simple algorithms:
    - Simple statement, If/else statements, Loops, Consecutive statement

Linked Lists

- In an array
  - the items are placed in consecutive places in the memory
    - Some operations in the middle of the list are costly
      - Insert
      - Delete
    - Static structure with a fixed total size
    - How much does it take to add an item at the end?

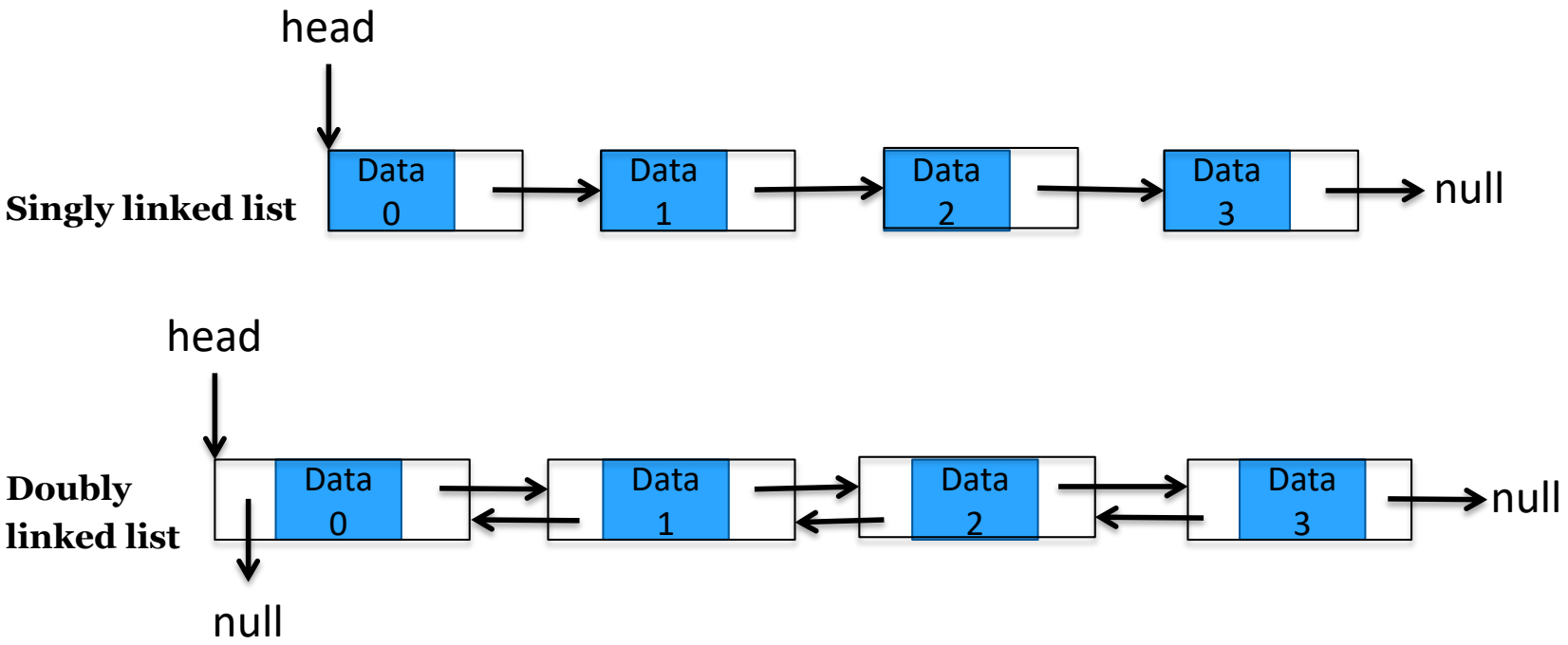


    - Insert at the end takes O(1)
    - But when there is not enough room, it takes O(n)

      because the whole array needs to be moved to a larger space

Linked Lists

- A dynamic data structure for representing a list, in which each item of the list is stored in a separate object called *Node*
- Nodes consist of an item and a reference to the next Node.
- We need a reference to the first node called head

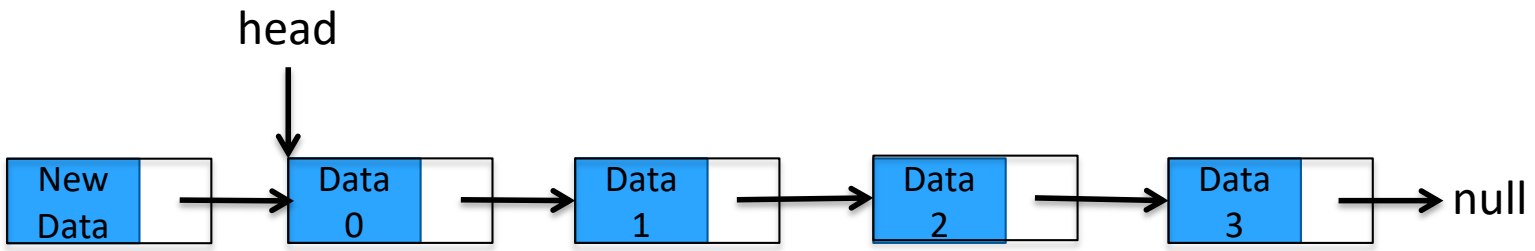


Node Struct and Methods

We need a structure for nodes and a number of methods

```
Struct Node{
    Type data;
    Pointer nextNode;
}
Class LinkedList{
   Pointer head;
// methods including insert, delete, search and traverse
   InsertAtFront(newData);
   Pointer Search(item);
   InsertAfter(newData, itemBefore);
 }
```

InsertAtFront method

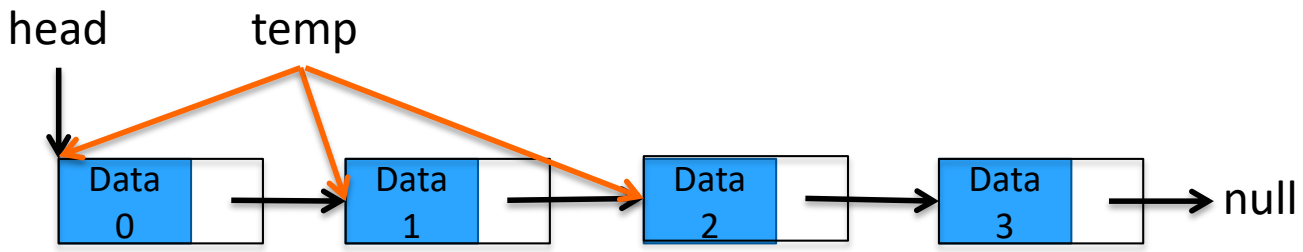



```
InsertAtFront(newData){
        newNode= new Node()
        newNode.data= newData
        newNode.nextNode= head
        head =  addressOf(newNode)
}
```

Complexity?

Takes O(1) time steps

Search method

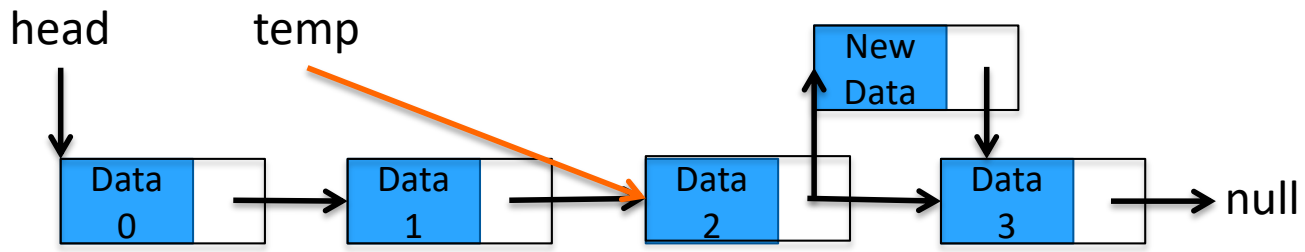



```
pointer Search(item){
        temp=head
        while(temp!=null && temp.data!=item)
                temp= temp.nextNode
        return temp
}
```

Complexity?

Takes O(n) time steps

InsertAfter method

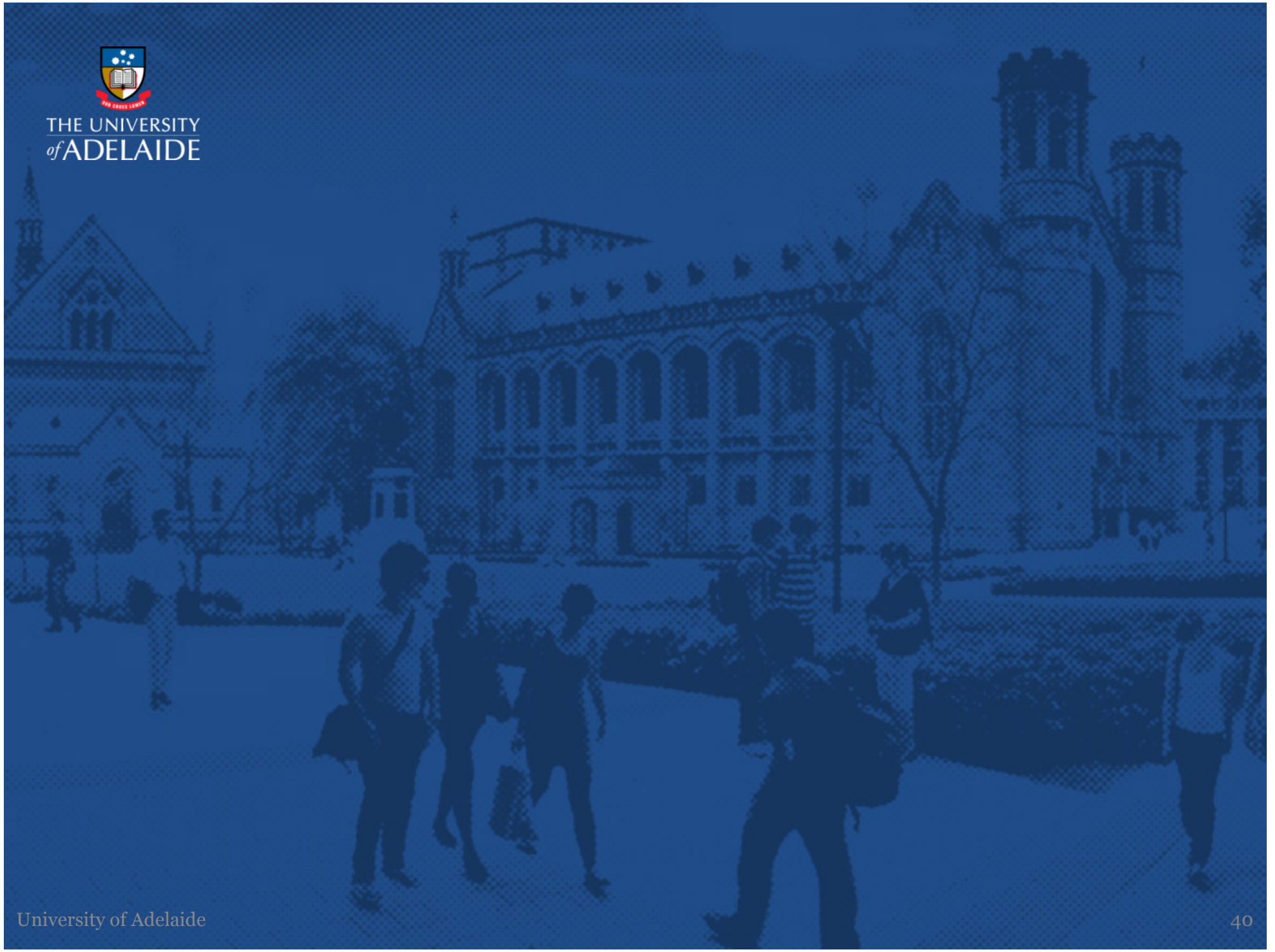



```
InsertAfter(newData, itemBefore){
        temp=Search(itemBefore)
        if (temp=null)
                error: item not found
        newNode= new Node()
        newNode.data= newData
        newNode.nextNode= temp.nextNode
        temp.nextNode=  addressOf(newNode)
}
```

Takes O(n) time steps

Summary on Linked lists

- We learned situations where array is not a good choice for representing lists
- We defined linked lists and learned a few methods for doing operations on linked lists
- Arrays:
  - Add at the end if enough space: O(1)
  - Due to fixed size, adding a new item at the end may take O(n)
  - Direct access to items by index number :O(1)
  - Shifts data when an item is added in the middle of the list or deleted from it: O(n)
- Linked Lists:
  - Dynamically grows or shrinks: add and remove take O(1)
  - No direct access by index number; Links should be followed: O(n)
  - Adding and removing items from the middle of the list include search: O(n), but not as costly as shifting the data


THE UNIVERSITY
of ADELAIDE