Week 2: Analysis of Algorithms, Dynamic Data Structures

Analysis of Algorithms

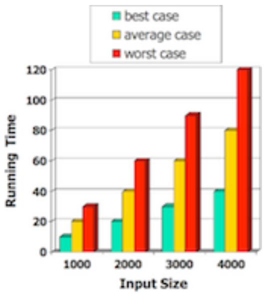
Running Time

An algorithm is a step-by-step procedure

- for solving a problem
- in a finite amount of time

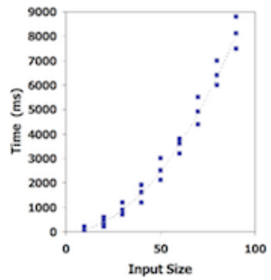
Most algorithms map input to output

- running time typically grows with input size
- *average time* often difficult to determine
- Focus on *worst case* running time
  - easier to analyse
  - crucial to many applications: finance, robotics, games, …



Empirical Analysis

1. Write program that implements an algorithm
2. Run program with inputs of varying size and composition
3. Measure the actual running time
4. Plot the results



Limitations:

- requires to implement the algorithm, which may be difficult
- results may not be indicative of running time on other inputs
- same hardware and operating system must be used in order to compare two algorithms

---

Theoretical Analysis

- Uses high-level description of the algorithm instead of implementation ("pseudocode")
- Characterises running time as a function of the input size, *n*
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

---

Pseudocode

Example: Find maximal element in an array

```
arrayMax(A):
|  Input  array A of n integers
|  Output maximum element of A
|
|  currentMax=A[0]
|  for all i=1..n-1 do
|  |  if A[i]>currentMax then
|  |     currentMax=A[i]
|  |  end if
|  end for
|  return currentMax
```

... Pseudocode

Control flow

- **if** … **then** … [**else**] … **end if**
- **while .. do** … **end while**
  repeat … **until**
  for [all][each] .. do … **end for**

Function declaration

- f(arguments):
  Input …
  Output …
  …

Expressions

- = assignment
- = equality testing
- n^2 superscripts and other mathematical formatting allowed
- swap A[i] and A[j] verbal descriptions of *simple* operations allowed

... Pseudocode

- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Exercise #1: Pseudocode

Formulate the following verbal description in pseudocode:

To reverse the order of the elements on a stack S with the help of a queue:

1. *In the first phase, pop one element after the other from S and enqueue it in queue Q until the stack is empty.*
2. *In the second phase, iteratively dequeue all the elements from Q and push them onto the stack.*

As a result, all the elements are now in reversed order on S.

Sample solution:

```
while S is not empty do
   pop e from S, enqueue e into Q
end while
while Q is not empty do
   dequeue e from Q, push e onto S
end while
```

Exercise #2: Pseudocode

Implement the following pseudocode instructions in C

1. `A` is an array of `int`s

   ```
   ...
   swap A[i] and A[j]
   ...
   ```

2. `S` is a stack

   ```
   ...
   swap the top two elements on S
   ...
   ```

1. ```
   int temp = A[i];
   A[i] = A[j];
   A[j] = temp;
   ```

2. ```
   x = StackPop(S);
   y = StackPop(S);
   StackPush(S, x);
   StackPush(S, y);
   ```

The following pseudocode instruction is problematic. Why?

```
...
swap the two elements at the front of queue Q
...
```

The Abstract RAM Model

RAM = Random Access Machine

- A CPU (central processing unit)
- A potentially unbounded bank of memory cells
  - each of which can hold an arbitrary number, or character
- Memory cells are numbered, and accessing any one of them takes CPU time

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent of the programming language
- Exact definition not important (we will shortly see why)
- Assumed to take a constant amount of time in the RAM model

Examples:

- evaluating an expression
- indexing into an array
- calling/returning from a function

Counting Primitive Operations

By inspecting the pseudocode …

- we can determine the maximum number of primitive operations executed by an algorithm
- as a function of the input size

Example:

```
arrayMax(A):
|  Input  array A of n integers
|  Output maximum element of A
|
|  currentMax=A[0]                 1
|  for all i=1..n-1 do             n+(n-1)
|  |  if A[i]>currentMax then      2(n-1)
|  |     currentMax=A[i]           n-1
|  |  end if
|  end for
|  return currentMax               1
                                   -------
                           Total   5n-2
```

Estimating Running Times

Algorithm `arrayMax` requires $5n - 2$ primitive operations in the *worst* case

- *best* case requires $4n - 1$ operations (why?)

Define:

- a … time taken by the fastest primitive operation
- b … time taken by the slowest primitive operation

Let $T(n)$ be worst-case time of `arrayMax`. Then

$a \cdot (5n - 2) \leq T(n) \leq b \cdot (5n - 2)$

Hence, the running time $T(n)$ is bound by two linear functions

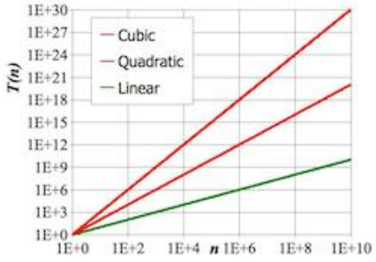
... Estimating Running Times

Seven commonly encountered functions for algorithm analysis

- Constant ≅ 1
- Logarithmic ≅ $\log n$
- Linear ≅ n
- N-Log-N ≅ $n \log n$
- Quadratic ≅ n^2
- Cubic ≅ n^3
- Exponential ≅ 2^n

... Estimating Running Times

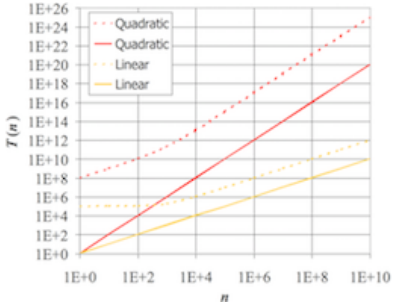
In a log-log chart, the slope of the line corresponds to the growth rate of the function



... Estimating Running Times

The growth rate is not affected by constant factors or lower-order terms

- Examples:
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function



... Estimating Running Times

Changing the hardware/software environment

- affects $T(n)$ by a constant factor
- but does not alter the growth rate of $T(n)$

⇒ *Linear* growth rate of the running time $T(n)$ is an intrinsic property of algorithm `arrayMax`

Exercise #3: Estimating running times

Determine the number of primitive operations

```
matrixProduct(A,B):
|  Input  n×n matrices A, B
|  Output n×n matrix A·B
|
|  for all i=1..n do
|  |  for all j=1..n do
|  |  |  C[i,j]=0
|  |  |  for all k=1..n do
|  |  |     C[i,j]=C[i,j]+A[i,k]·B[k,j]
|  |  |  end for
|  |  end for
|  end for
|  return C
```

```
matrixProduct(A,B):
|  Input  n×n matrices A, B
|  Output n×n matrix A·B
|
|  for all i=1..n do                           2n+1
|  |  for all j=1..n do                        n(2n+1)
|  |  |  C[i,j]=0                              n^2
|  |  |  for all k=1..n do                     n^2(2n+1)
|  |  |     C[i,j]=C[i,j]+A[i,k]·B[k,j]        n^3·4
|  |  |  end for
|  |  end for
|  end for
|  return C                                    1
                                               ------------
                                       Total   6n^3+4n^2+3n+2
```

Big-Oh

Big-Oh Notation

27/180

Given functions f(n) and g(n), we say that

$$f(n) \in O(g(n))$$

if there are positive constants c and n_0 such that

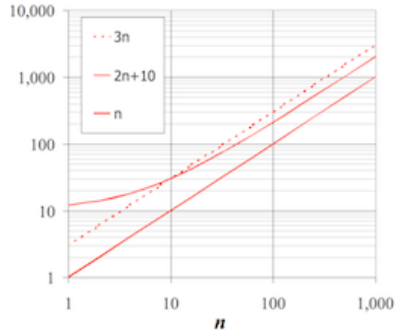
$$f(n) \le c \cdot g(n) \quad \forall n \ge n_0$$

Hence: *O(g(n)) is the set of all functions that do not grow faster than g(n)*

... Big-Oh Notation

28/180

Example: function $2n + 10$ is in O(n)

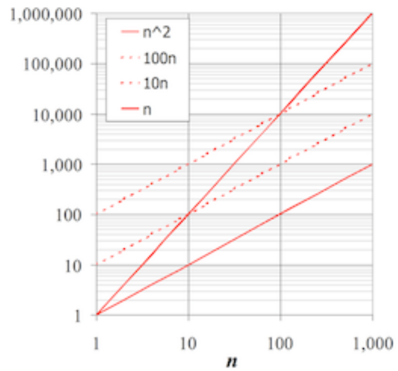


- $2n+10 \le c \cdot n$
  $\Rightarrow$ $(c-2)n \ge 10$
  $\Rightarrow$ $n \ge 10/(c-2)$
- pick c=3 and n_0=10

... Big-Oh Notation

29/180

Example: function n^2 is not in O(n)



- $n^2 \le c \cdot n$
  $\Rightarrow$ $n \le c$
- inequality cannot be satisfied since c must be a constant

Exercise #4: Big-Oh

30/180

Show that

1. 7n-2 is in O(n)
2. $3n^3 + 20n^2 + 5$ is in O(n^3)
3. $3 \cdot \log_2 n + 5$ is in O($\log_2 n$)

---

1. $7n-2 \in O(n)$
   need $c>0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$
   $\Rightarrow$ true for $c=7$ and $n_0=1$
2. $3n^3 + 20n^2 + 5 \in O(n^3)$
   need $c>0$ and $n_0 \geq 1$ such that $3n^3+20n^2+5 \leq c \cdot n^3$ for $n \geq n_0$
   $\Rightarrow$ true for $c=4$ and $n_0=21$
3. $3 \cdot \log n + 5 \in O(\log n)$
   need $c>0$ and $n_0 \geq 1$ such that $3 \cdot \log_2 n+5 \leq c \cdot \log n$ for $n \geq n_0$
   $\Rightarrow$ true for $c=8$ and $n_0=2$ (can also choose $c=4$, along with which n_0?)

Big-Oh and Rate of Growth

32/180

- Big-Oh notation gives an upper bound on the growth rate of a function
  - "$f(n) \in O(g(n))$" means growth rate of f(n) no more than growth rate of g(n)
- use big-Oh to rank functions according to their rate of growth

	$f(n) \in O(g(n))$	$g(n) \in O(f(n))$
g(n) grows faster	yes	no
f(n) grows faster	no	yes
same order of growth	yes	yes

Big-Oh Rules

33/180

- If f(n) is a polynomial of degree d $\Rightarrow$ f(n) is $O(n^d)$
  - lower-order terms are ignored
  - constant factors are ignored
- Use the smallest possible class of functions
  - say "2n is O(n)" instead of "2n is $O(n^2)$"
    - but keep in mind that, 2n is in $O(n^2)$, $O(n^3)$, …
- Use the simplest expression of the class
  - say "3n + 5 is O(n)" instead of "3n + 5 is O(3n)"

Exercise #5: Big-Oh

34/180

Show that $\sum_{i=1}^{n} i = 1 + 2 + 3 + \dots + n$ is $O(n^2)$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

which is $O(n^2)$

Asymptotic Analysis of Algorithms

36/180

Asymptotic analysis of algorithms determines running time in big-Oh notation:

- find worst-case number of primitive operations as a function of input size
- express this function using big-Oh notation

Example:

- algorithm `arrayMax` executes at most 5n – 2 primitive operations
  $\Rightarrow$ algorithm `arrayMax` "*runs in O(n) time*"

Constant factors and lower-order terms eventually dropped
$\Rightarrow$ can disregard them when counting primitive operations

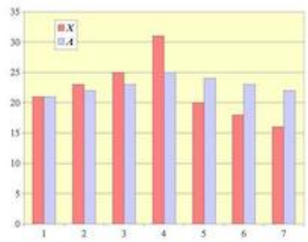
---

Example: Computing Prefix Averages

37/180

- The *i-th prefix average* of an array X is the average of the first i elements:

A[i] = (X[0] + X[1] + … + X[i]) / (i+1)



NB. computing the array A of prefix averages of another array X has applications in financial analysis

---

... Example: Computing Prefix Averages

38/180

A *quadratic* algorithm to compute prefix averages:

```
prefixAverages1(X):
|  Input  array X of n integers
|  Output array A of prefix averages of X
|
|  for all i=0..n-1 do          O(n)
|  |  s=X[0]                    O(n)
```

```
|   |   for all j=1..i do          O(n^2)
|   |      s=s+X[j]                O(n^2)
|   |   end for
|   |   A[i]=s/(i+1)               O(n)
|   end for
|   return A                       O(1)
```

$$2 \cdot O(n^2) + 3 \cdot O(n) + O(1) = O(n^2)$$

⇒ *Time complexity* of algorithm `prefixAverages1` is $O(n^2)$

... Example: Computing Prefix Averages

39/180

The following algorithm computes prefix averages by keeping a running sum:

```
prefixAverages2(X):
|  Input  array X of n integers
|  Output array A of prefix averages of X
|
|  s=0
|  for all i=0..n-1 do          O(n)
|     s=s+X[i]                  O(n)
|     A[i]=s/(i+1)              O(n)
|  end for
|  return A                     O(1)
```

Thus, `prefixAverages2` is $O(n)$

Example: Binary Search

40/180

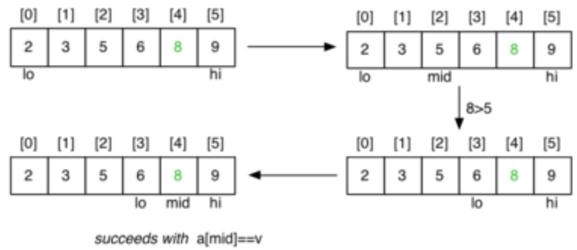
The following recursive algorithm searches for a value in a *sorted* array:

```
search(v,a,lo,hi):
|  Input  value v
|         array a[lo..hi] of values
|  Output true if v in a[lo..hi]
|         false otherwise
|
|  if lo>hi then return false
|  mid=(lo+hi)/2
|  if a[mid]=v then
|     return true
|  else if a[mid]<v then
|     return search(v,a,mid+1,hi)
|  else
|     return search(v,a,lo,mid-1)
|  end if
```

... Example: Binary Search

41/180

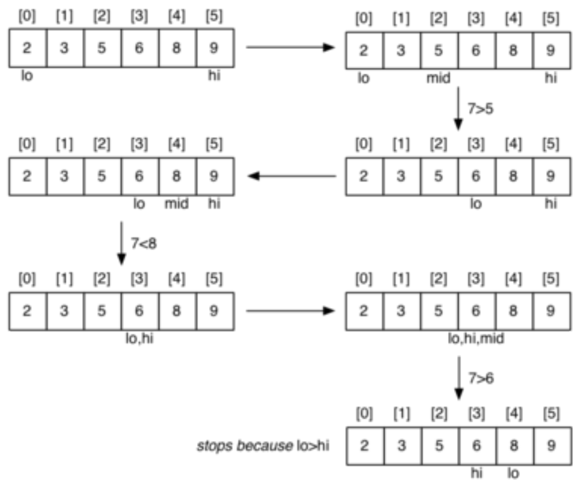
Successful search for a value of 8:



... Example: Binary Search

42/180

Unsuccessful search for a value of 7:



... Example: Binary Search

43/180

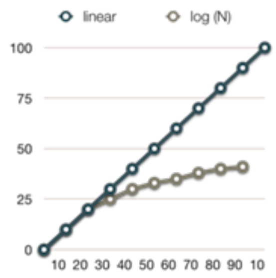
Cost analysis:

- C_i = #calls to `search()` for array of length i
- for best case, $C_n = 1$
- for `a[i..j]`, `j<i` (length=0)
  - $C_0 = 0$
- for `a[i..j]`, `i≤j` (length=n)
  - $C_n = 1 + C_{n/2} \;\Rightarrow C_n = \log_2 n$

Thus, binary search is $O(\log_2 n)$ or simply $O(\log n)$ (why?)

... Example: Binary Search

Why logarithmic complexity is good:



Math Needed for Complexity Analysis

- Logarithms
  - $\log_b (xy) = \log_b x + \log_b y$
  - $\log_b (x/y) = \log_b x - \log_b y$
  - $\log_b x^a = a \log_b x$
  - $\log_a x = \log_b x \cdot (\log_c b / \log_c a)$
- Exponentials
  - $a^{(b+c)} = a^b a^c$
  - $a^{bc} = (a^b)^c$
  - $a^b / a^c = a^{(b-c)}$
  - $b = a^{\log_a b}$
  - $b^c = a^{c \cdot \log_a b}$
- Proof techniques
- Summation (addition of sequences of numbers)
- Basic probability (for average case analysis, randomised algorithms)

Exercise #6: Analysis of Algorithms

What is the complexity of the following algorithm?

```
enqueue(Q,Elem):
|  Input  queue Q, element Elem
|  Output Q with Elem added at the end
|
|  Q.top=Q.top+1
|  for all i=Q.top down to 1 do
|     Q[i]=Q[i-1]
|  end for
|  Q[0]=Elem
|  return Q
```

Answer: O(|Q|)

Exercise #7: Analysis of Algorithms

What is the complexity of the following algorithm?

```
insertionSort(A):
|  Input array A[0..n-1] of n elements
|
|  for all i=1..n-1 do
|  |  element=A[i], j=i-1
|  |  while j≥0 and A[j]>element do
|  |     A[j+1]=A[j]
|  |     j=j-1
|  |  end while
|  |  A[j+1]=element
|  end for
```

Answer: $O(n^2)$

Best known sorting algorithms are $O(n \cdot \log n)$. Example: Quicksort (→ week 5)

Exercise #8: Analysis of Algorithms

What is the complexity of the following algorithm?

```
binaryConversion(n):
|  Input  positive integer n
|  Output binary representation of n on a stack
|
|  create empty stack S
|  while n>0 do
|  |  push (n mod 2) onto S
|  |  n=⌊n/2⌋
|  end while
|  return S
```

Assume that creating a stack and pushing an element both are O(1) operations ("constant")

Answer: O(log n)

Relatives of Big-Oh

big-Omega

- $f(n) \in \Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$$

big-Theta

- $f(n) \in \Theta(g(n))$ if there are constants $c',c'' > 0$ and an integer constant $n_0 \geq 1$ such that

$$c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n) \quad \forall n \geq n_0$$

... Relatives of Big-Oh

53/180

- $f(n)$ belongs to $O(g(n))$ if $f(n)$ is asymptotically *less than or equal* to $g(n)$
- $f(n)$ belongs to $\Omega(g(n))$ if $f(n)$ is asymptotically *greater than or equal* to $g(n)$
- $f(n)$ belongs to $\Theta(g(n))$ if $f(n)$ is asymptotically *equal* to $g(n)$

... Relatives of Big-Oh

54/180

Examples:

- $\frac{1}{4}n^2 \in \Omega(n^2)$
  - need $c > 0$ and $n_0 \geq 1$ such that $\frac{1}{4}n^2 \geq c \cdot n^2$ for $n \geq n_0$
  - let $c=\frac{1}{4}$ and $n_0=1$
- $\frac{1}{4}n^2 \in \Omega(n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $\frac{1}{4}n^2 \geq c \cdot n$ for $n \geq n_0$
  - let $c=1$ and $n_0=4$
- $\frac{1}{4}n^2 \in \Theta(n^2)$
  - since $\frac{1}{4}n^2$ belongs to $\Omega(n^2)$ and $O(n^2)$
- $\frac{1}{4}n^2 \notin \Theta(n)$
  - since $\frac{1}{4}n^2$ does not belong to $O(n)$

Complexity Analysis: Arrays vs. Linked Lists

Static/Dynamic Sequences

56/180

Previously we have used an *array* to implement a stack

- fixed size collection of homogeneous elements
- can be accessed via index or via "moving" pointer

The "fixed size" aspect is a potential problem:

- how big to make the (dynamic) array? (big … just in case)
- what to do if it fills up?

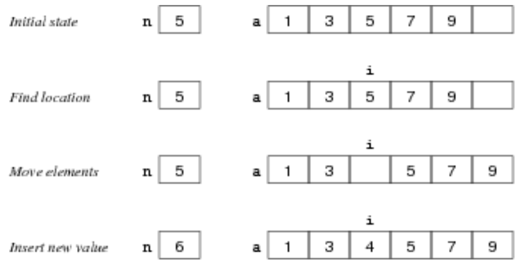
The rigid sequence is another problems:

- inserting/deleting an item in middle of array

... Static/Dynamic Sequences

57/180

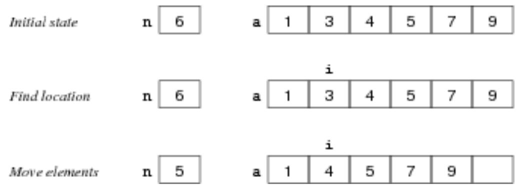
Inserting a value (*4*) into a sorted array `a` with `n` elements:



... Static/Dynamic Sequences

58/180

Deleting a value (*3*) from a sorted array `a` with `n` elements:

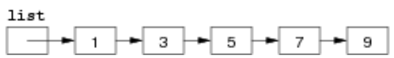


... Static/Dynamic Sequences

59/180

The problems with using arrays can be solved by

- allocating elements individually
- linking them together as a "chain"



Benefits:

- insertion/deletion have minimal effect on list overall
- only use as much space as needed for values

Self-referential Structures

60/180

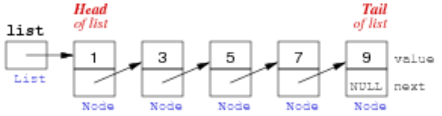
To realise a "chain of elements", need a *node* containing

- a value

- a link to the next node

To represent a chained (linked) *list* of nodes:

- we need a *pointer* to the first node
- each node contains a pointer to the next node
- the `next` pointer in the last node is `NULL`



... Self-referential Structures

Linked lists are more flexible than arrays:

- values do not have to be adjacent in memory
- values can be rearranged simply by altering pointers
- the number of values can change dynamically
- values can be added or removed in any order

Disadvantages:

- it is not difficult to get pointer manipulations wrong
- each value also requires storage for `next` pointer

... Self-referential Structures

Create a new list node:

```
makeNode(v)
|  Input   value v
|  Output  new linked list node with value v
|
|  new.value=v      // initialise data
|  new.next=NULL    // initialise link to next node
|  return new       // return pointer to new node
```

Exercise #9: Creating a Linked List

Write pseudocode that uses **`makeNode(v)`** to create a linked list of three nodes with values 1, 42 and 9024.

```
mylist=makeNode(1)
mylist.next=makeNode(42)
(mylist.next).next=makeNode(9024)
```

Iteration over Linked Lists

When manipulating list elements

- typically have pointer `p` to current node
- to access the data in current node: `p.value`
- to get pointer to next node: `p.next`

To iterate over a linked list:

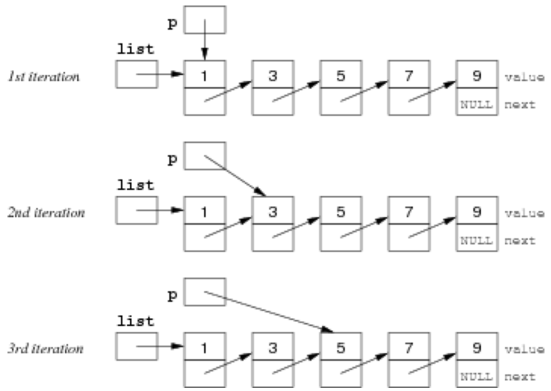
- set `p` to point at first node (head)
- examine node pointed to by `p`
- change `p` to point to next node
- stop when `p` reaches end of list (`NULL`)

... Iteration over Linked Lists

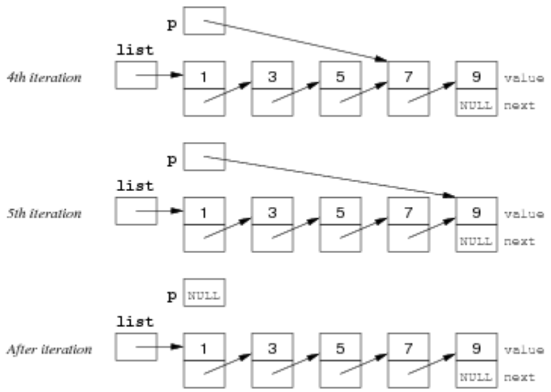
Standard method for scanning all elements in a linked list:

```
list  // pointer to first Node in list
p     // pointer to "current" Node in list

p=list
while p≠NULL do
|  … do something with p.value …
|  p=p.next
end while
```

... Iteration over Linked Lists



... Iteration over Linked Lists



... Iteration over Linked Lists

Check if list contains an element:

```
inLL(L,d):
|  Input  linked list L, value d
|  Output true if d in list, false otherwise
|
|  p=L
|  while p≠NULL do
|  |  if p.value=d then   // element found
|  |     return true
|  |  end if
|  |  p=p.next
|  end while
|  return false          // element not in list
```

Time complexity: O(|L|)

... Iteration over Linked Lists

Print all elements:

```
showLL(L):
|  Input linked list L
|
|  p=L
|  while p≠NULL do
|     print p.value
|     p=p.next
|  end while
```

Time complexity: O(|L|)

Exercise #10: Traversing a linked list

What does this code do?

```
1  p=list
2  while p≠NULL do
3  |  print p.value
4  |  if p.next≠NULL then
5  |     p=p.next.next
6  |  else
7  |     p=NULL
8  |  end if
9  end while
```

What is the purpose of the conditional statement in line 4?

Every second list element is printed.

If `p` happens to be the last element in the list, then `p.next.next` does not exist.
The if-statement ensures that we do not attempt to assign an undefined value to pointer `p` in line 5.

Exercise #11: Traversing a linked list

Rewrite **`showLL(L)`** as a recursive function.

```
showLL(L):
|  Input linked list L
|
|  if L≠NULL do
|     print L.value
|     showLL(L.next)
|  end if
```

Modifying a Linked List

Insert a new element at the beginning:

```
insertLL(L,d):
|  Input  linked list L, value d
|  Output L with d prepended to the list
|
|  new=makeNode(d)   // create new list element
|  new.next=L        // link to beginning of list
|  return new        // new element is new head
```

Time complexity: O(1)

... Modifying a Linked List

Delete the *first* element:

```
deleteHead(L):
|  Input  non-empty linked list L
|  Output L with head deleted
|
|  return L.next    // move to second element
```

Time complexity: O(1)

Delete a *specific* element (recursive version):

```
deleteLL(L,d):
|  Input  linked list L, value d
|  Output L with element d deleted
|
|  if L=NULL then              // element not in list
|     return L
|  else if L.value=d then      // d found at front
|     return deleteHead(L)     // delete first element
|  else                        // delete element in tail list
|     L.next=deleteLL(L.next,d)
|  end if
|  return L
```

Time complexity: O(|L|)

Exercise #12: Implementing a Queue as a Linked List

Develop a datastructure for a queue based on linked lists such that …

- enqueuing an element takes constant time
- dequeuing an element takes constant time

Use pointers to both ends

Dequeue from the front …

```
dequeue(Q):
|  Input  non-empty queue Q
|  Output front element d, dequeued from Q
|
|  d=Q.front.value        // first element in the list
|  Q.front=Q.front.next   // move to second element
|  return d
```

Enqueue at the rear …

```
enqueue(Q,d):
|  Input queue Q
|
|  new=makeNode(d)       // create new list element
|  Q.rear.next=new       // add to end of list
|  Q.rear=new            // link to new end of list
```

Comparison Array vs. Linked List

Complexity of operations, *n* elements

	array	linked list
insert/delete at beginning	$O(n)$	$O(1)$
insert/delete at end	$O(1)$	$O(1)$ ("doubly-linked" list, with pointer to rear)
insert/delete at middle	$O(n)$	$O(n)$
find an element	$O(n)$ ($O(\log n)$, if array is sorted)	$O(n)$
index a specific element	$O(1)$	$O(n)$

Complexity Classes

Complexity Classes

Problems in Computer Science …

- some have *polynomial* worst-case performance (e.g. n^2)
- some have *exponential* worst-case performance (e.g. 2^n)

Classes of problems:

- *P* = problems for which an algorithm can compute answer in polynomial time
- *NP* = includes problems for which no *P* algorithm is known

Beware: NP stands for "nondeterministic, polynomial time (on a theoretical *Turing Machine*)"

... Complexity Classes

Computer Science jargon for difficulty:

- tractable … have a polynomial-time algorithm (useful in practice)
- intractable … no tractable algorithm is known (feasible only for small n)
- non-computable … no algorithm can exist

Computational complexity theory deals with different degrees of intractability

Generate and Test

83/180

In scenarios where

- it is simple to test whether a given state is a solution
- it is easy to generate new states (preferably likely solutions)

then a *generate and test* strategy can be used.

It is necessary that states are generated systematically

- so that we are guaranteed to find a solution, or know that none exists
  - some randomised algorithms do not require this, however (more on this later in this course)

... Generate and Test

84/180

Simple example: checking whether an integer n is prime

- generate/test all possible factors of n
- if none of them pass the test $\Rightarrow$ n is prime

Generation is straightforward:

- produce a sequence of all numbers from 2 to n-1

Testing is also straightforward:

- check whether next number divides n exactly

... Generate and Test

85/180

Function for primality checking:

```
isPrime(n):
|  Input  natural number n
|  Output true if n prime, false otherwise
|
|  for all i=2..n-1 do       // generate
|  |  if n mod i = 0 then    // test
|  |     return false        // i is a divisor => n is not prime
|  |  end if
|  end for
|  return true               // no divisor => n is prime
```

Complexity of `isPrime` is O(n)

Can be optimised: check only numbers between 2 and $\lfloor\sqrt{n}\rfloor$ $\Rightarrow$ O($\sqrt{n}$)

Example: Subset Sum

86/180

Problem to solve ...

Is there a subset S of these numbers with $\Sigma_{x\in S}\, x = 1000$?

```
 34,  38,  39,  43,  55,  66,  67,  84,  85,  91,
101, 117, 128, 138, 165, 168, 169, 182, 184, 186,
234, 238, 241, 276, 279, 288, 386, 387, 388, 389
```

General problem:

- given n arbitrary integers and a target sum k
- is there a subset that adds up to exactly k?

... Example: Subset Sum

87/180

Generate and test approach:

```
subsetsum(A,k):
|  Input  set A of n integers, target sum k
|  Output true if Σx∈Sx=k for some S⊆A
|         false otherwise
|
|  for each subset B⊆A do
|  |  if Σb∈Bb=k then
|  |     return true
|  |  end if
|  end for
|  return false
```

- How many subsets are there of n elements?
- How could we generate them?

... Example: Subset Sum

88/180

Given: a set of `n` distinct integers in an array `A` …

- produce all subsets of these integers

A method to generate subsets:

- represent sets as n bits (e.g. n=4, `0000`, `0011`, `1111` etc.)
- bit i represents the i^{th} input number
- if bit i is set to 1, then `A[i]` is in the subset
- if bit i is set to 0, then `A[i]` is not in the subset

- e.g. if `A[]=={1,2,3,5}` then `0011` represents `{1,2}`

... Example: Subset Sum

Algorithm:

```
subsetsum1(A,k):
|  Input   set A of n integers, target sum k
|  Output  true if Σx∈S x=k for some S⊆A
|          false otherwise
|
|  for s=0..2^n-1 do
|  |  if k = Σ(ith bit of s is 1) A[i] then
|  |     return true
|  |  end if
|  end for
|  return false
```

Obviously, `subsetsum1` is $O(2^n)$

... Example: Subset Sum

Alternative approach …

`subsetsum2(A,n,k)`
(returns true if any subset of A[0..*n*-1] sums to *k*; returns false otherwise)

- if the n^{th} value A[*n*-1] is part of a solution …
  - then the first *n*-1 values must sum to *k* – A[*n*-1]
- if the n^{th} value is not part of a solution …
  - then the first *n*-1 values must sum to *k*
- base cases: *k*=0 (solved by {}); *n*=0 (unsolvable if *k*>0)

```
subsetsum2(A,n,k):
|  Input   array A, index n, target sum k
|  Output  true if some subset of A[0..n-1] sums up to k
|          false otherwise
|
|  if k=0 then
|     return true   // empty set solves this
|  else if n=0 then
|     return false  // no elements => no sums
|  else
|     return subsetsum(A,n-1,k-A[n-1]) or subsetsum(A,n-1,k)
|  end if
```

... Example: Subset Sum

Cost analysis:

- C_i = #calls to `subsetsum2()` for array of length i
- for worst case,
  - $C_1 = 2$
  - $C_n = 2 + 2 \cdot C_{n-1} \Rightarrow C_n \cong 2^n$

Thus, `subsetsum2` also is $O(2^n)$

... Example: Subset Sum

Subset Sum is typical member of the class of *NP-complete problems*

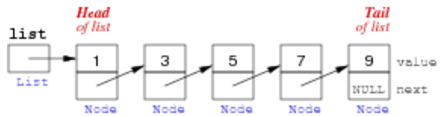
- intractable … only algorithms with exponential performance are known
  - increase input size by 1, double the execution time
  - increase input size by 100, it takes 2^{100} = 1,267,650,600,228,229,401,496,703,205,376 times as long to execute
- but if you can find a polynomial algorithm for Subset Sum, then any other *NP*-complete problem becomes *P* …

Pointers

Pointers

Reminder: In a *linked list* …

- each node contains a pointer to the next node
- the number of values can change dynamically



Benefits:

- insertion/deletion have minimal effect on list overall
- only use as much space as needed for values

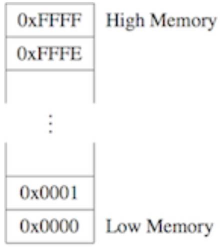
In C, linked lists are implemented using *pointers* and *dynamic memory allocation*

Memory

Reminder: Computer memory … large array of consecutive data cells or *bytes*

- 1 byte = 8 bits = `0x00` … `0xFF`
- `char` … 1 byte   `int,float` … 4 bytes   `double` … 8 bytes

Memory addresses shown in Hexadecimal notation



When a variable is declared, the operating system finds a place in memory to store the appropriate number of bytes.

If we declare a variable called `k` …

- the place where `k` is stored is denoted by `&k`
- also called the address of k

... Memory

Example:

```
int k;
int m;

printf("address of k is %p\n", &k);
printf("address of m is %p\n", &m);
```

```
address of k is BFFFFB80
address of m is BFFFFB84
```

This means that

- `k` occupies the four bytes from `BFFFFB80` to `BFFFFB83`
- `m` occupies the four bytes from `BFFFFB84` to `BFFFFB87`

Note the use of **`%p`** as placeholder for an address ("pointer" value)

Application: Input Using `scanf()`

Standard I/O function **`scanf()`** to read formatted input

- `scanf()` uses a format string like `printf()`
- requires the *address* of a variable as argument
- use **`%d`** to read an integer value (**`%f`** for float, **`%lf`** for `double`)

```
#include <stdio.h>
…
int answer;
printf("Enter your answer: ");
scanf("%d", &answer);
```

- `scanf()` returns a value — the number of items read
  - use this value to determine if `scanf()` successfully read a number
    - `scanf()` could fail e.g. if the user enters letters

Exercise #13: Using `scanf`

Write a program that

- asks the user for a number
- checks that it is positive
- applies Collatz's process (Exercise 3a, Problem set week 1) to the number

---

```
#include <stdio.h>

void collatz(int n) {
   printf("%d\n", n);
   while (n != 1) {
      if (n % 2 == 0)
         n = n / 2;
      else
         n = 3*n + 1;
      printf("%d\n", n);
   }
}

int main(void) {
   int n;
   printf("Enter a positive number: ");
   if (scanf("%d", &n) == 1 && (n > 0))  /* test if scanf successful
                                            and returns positive number */
      collatz(n);
   return 0;
}
```

Pointers

A *pointer* …

- is a special type of variable
- storing the address (memory location) of another variable

A pointer occupies space in memory, just like any other variable of a certain type

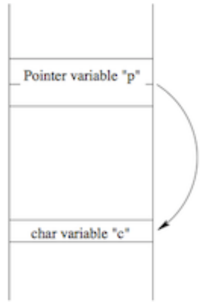
The number of memory cells needed for a pointer depends on the computer's architecture:

- First generation desktop computer, or hand-held device with only 64KB of addressable memory:
  - 2 memory cells (i.e. 16 bits) to hold any address from `0x0000` to `0xFFFF` (= 65535)
- Desktop machine with 4GB of addressable memory
  - 4 memory cells (i.e. 32 bits) to hold any address from `0x00000000` to `0xFFFFFFFF` (= 4294967295)
- Modern 64-bit computer
  - 8 memory cells (can address 2^{64} bytes, but in practice the amount of memory is limited by the CPU)

... Pointers

Suppose we have a pointer **`p`** that "points to" a `char` variable `c`.

Assuming that the pointer **p** requires 2 bytes to store the address of **c**, here is what the memory map might look like:



... Pointers

106/180

Now that we have assigned to p the address of variable `c` ...

- need to be able to reference the data in that memory location

Operator `*` is used to access the object the pointer points to

- e.g. to change the value of `c` using the pointer `p`:

```
*p = 'T';  // sets the value of c to 'T'
```

The `*` operator is sometimes described as "*dereferencing*" the pointer, to access the underlying variable

---

... Pointers

107/180

Things to note:

- all pointers constrained to point to a particular type of object

```
// a potential pointer to any object of type char
char *s;

// a potential pointer to any object of type int
int *p;
```

- if pointer `p` is pointing to an integer variable `x`
  $\Rightarrow$ `*p` can occur in any context that `x` could

Examples of Pointers

108/180

```
int *p; int *q; // this is how pointers are declared
int a[5];
int x = 10, y;
 p = &x;       // p now points to x
*p = 20;       // whatever p points to is now equal to 20
 y = *p;       // y is now equal to whatever p points to
 p = &a[2];    // p points to an element of array a[]
 q = p;        // q and p now point to the same thing
```

Exercise #14: Pointers

109/180

What is the output of the following program?

```
#include <stdio.h>

int main(void) {
   int *ptr1, *ptr2;
   int i = 10, j = 20;

   ptr1 = &i;
   ptr2 = &j;

   *ptr1 = *ptr1 + *ptr2;
   ptr2 = ptr1;
   *ptr2 = 2 * (*ptr2);
   printf("Val = %d\n", *ptr1 + *ptr2);
   return 0;
}
```

`Val = 120`

... Examples of Pointers

111/180

Can we write a function to "swap" two variables?

The *wrong* way:

```
void swap(int a, int b) {
   int temp = a;                      // only local "copies" of a and b will swap
   a = b;
   b = temp;
}

int main(void) {
   int a = 5, b = 7;
   swap(a, b);
   printf("a = %d, b = %d\n", a, b); // a and b still have their original values
   return 0;
}
```

... Examples of Pointers

In C, parameters are "call-by-value"

- changes made to the value of a parameter do not affect the original
- function `swap()` tries to swap the values of `a` and `b`, but fails because it only swaps the copies, not the "real" variables in `main()`

We can achieve "simulated call-by-reference" by passing pointers as parameters

- this allows the function to change the "actual" value of the variables

---

... Examples of Pointers

Can we write a function to "swap" two variables?

The *right* way:

```
void swap(int *p, int *q) {
   int temp = *p;                 // changes the actual values of a and b
   *p = *q;
   *q = temp;
}

int main(void) {
   int a = 5, b = 7;
   swap(&a, &b);
   printf("a = %d, b = %d\n", a, b);  // a and b now successfully swapped
   return 0;
}
```

Pointers and Arrays

When an array is declared, the elements of the array are guaranteed to be stored in consecutive memory locations:

```
int array[5];

for (i = 0; i < 5; i++) {
   printf("address of array[%d] is %p\n", i, &array[i]);
}
```

```
address of array[0] is BFFFFB60
address of array[1] is BFFFFB64
address of array[2] is BFFFFB68
address of array[3] is BFFFFB6C
address of array[4] is BFFFFB70
```

Pointer Arithmetic

A *pointer* variable holds a value which is an *address*.

C knows what type of object is being pointed to

- it knows the `sizeof` that object
- it can compute where the next/previous object is located

Example:

```
int a[6];   // assume array starts at address 0x1000
int *p;
p = &a[0];  // p contains 0x1000
p = p + 1;  // p now contains 0x1004
```

... Pointer Arithmetic

For a pointer declared as `T *p;` (where `T` is a type)

- if the pointer initially contains address `A`
  - executing `p = p + k;` (where `k` is a constant)
    - changes the value in `p` to `A + k*sizeof(T)`

The value of `k` can be positive or negative.

Example:

```
int a[6];   (addr 0x1000)       char s[10];   (addr 0x2000)
int *p;     (p == ?)           char *q;      (q == ?)
p = &a[0];  (p == 0x1000)       q = &s[0];    (q == 0x2000)
p = p + 2;  (p == 0x1008)       q++;          (q == 0x2001)
```

... Pointer Arithmetic

An alternative approach to iteration through an array:

- determine the address of the first element in the array
- determine the address of the last element in the array
- set a pointer variable to refer to the first element
- use pointer arithmetic to move from element to element
- terminate loop when address exceeds that of last element

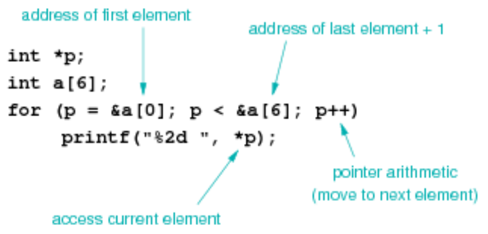
Example:

```
int a[6];
int *p;
p = &a[0];
while (p <= &a[5]) {
    printf("%2d ", *p);
    p++;
}
```

... Pointer Arithmetic

118/180

Pointer-based scan written in more typical style



Note: because of pointer/array connection `a[i] == *(a+i)` specifically, `a[0] == *a`

Arrays of Strings

119/180

One common type of pointer/array combination are the *command line arguments*

- These are 0 or more strings specified when program is run
- Suppose you have an excutable program named `seqq`. If you run this command in a terminal:

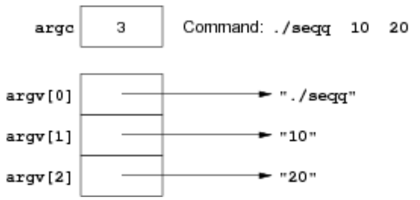
  ```
  prompt$ ./seqq 10 20
  ```

  then `seqq` will be given 2 command-line arguments: `"10", "20"`

... Arrays of Strings

120/180

```
prompt$ ./seqq 10 20
```



Each element of `argv[]` is

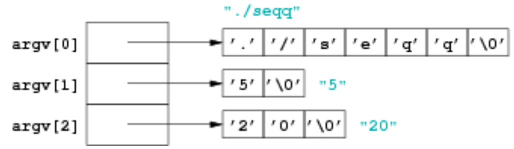
- a pointer to the start of a character array (`char *`)
  - containing a `\0`-terminated string

... Arrays of Strings

121/180

More detail on how `argv` is represented:

```
prompt$ ./seqq 5 20
```



Note the difference between `argv[i]` and `*argv[i]`:

- `argv[i]` is an address (== a string)
- `*argv[i]` is a single character (== `argv[i][0]`)

... Arrays of Strings

122/180

`main()` needs different prototype if you want to access command-line arguments:

```
int main(int argc, char *argv[]) { ...
```

- `argc` ... stores the number of command-line arguments + 1
  - `argc == 1` if no command-line arguments
- `argv[]` ... stores program name + command-line arguments
  - `argv[0]` always contains the program name
  - `argv[1],argv[2],`... are the command-line arguments if supplied

`<stdlib.h>` defines useful functions to convert strings:

- `atoi(char *s)` converts string to int
- `atof(char *s)` converts string to double (can also be assigned to `float` variable)

Exercise #15: Command Line Arguments

123/180

Write a program that

- checks for a single command line argument
  - if not, outputs a usage message and exits with failure
- converts this argument to a number and checks that it is positive
- applies Collatz's process (Exercise 3, Problem set week 1) to the number

---

```
#include <stdio.h>
#include <stdlib.h>

void collatz(int n) {
   ...
}
```

```
int main(int argc, char *argv[]) {
   if (argc != 2) {
      printf("Usage: %s number\n", argv[0]);
      return 1;
   }
   int n = atoi(argv[1]);
   if (n > 0)
      collatz(n);
   return 0;
}
```

... Arrays of Strings

125/180

`argv` can also be viewed as *double pointer* (a pointer to a pointer)

⇒ Alternative prototype for `main()`:

```
int main(int argc, char **argv) { ...
```

Can still use `argv[0]`, `argv[1]`, …

Pointers and Structures

126/180

Like any object, we can get the address of a `struct` via `&`.

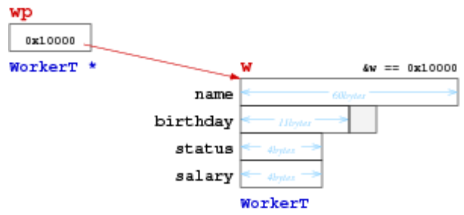
```
typedef char Date[11];  // e.g. "03-08-2017"
typedef struct {
    char  name[60];
    Date  birthday;
    int   status;      // e.g. 1 (≡ full time)
    float salary;
} WorkerT;

WorkerT w;  WorkerT *wp;
wp = &w;
// a problem ...
*wp.salary = 125000.00;
// does not have the same effect as
w.salary = 125000.00;
// because it is interpreted as
*(wp.salary) = 125000.00;

// to achieve the correct effect, we need
(*wp).salary = 125000.00;
// a simpler alternative is normally used in C
wp->salary = 125000.00;
```

Learn this well; we will frequently use it in this course.

... Pointers and Structures

127/180

Diagram of scenario from program above:



... Pointers and Structures

128/180

General principle …

If we have:

```
SomeStructType s;
SomeStructType *sp = &s;  // declare pointer and initialise to address of s
```

then the following are all equivalent:

`s.SomeElem`   `sp->SomeElem`   `(*sp).SomeElem`

C execution: Memory

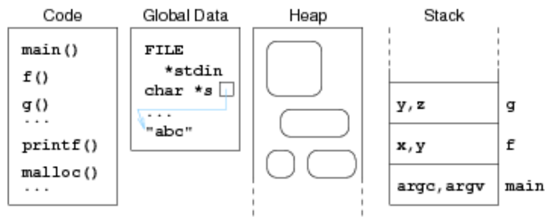
129/180

An executing C program partitions memory into:

- *code* … fixed-size, read-only region
    - contains the machine code instructions for the program
- *global data* … fixed-size
    - contain global variables (read-write) and constant strings (read-only)
- *heap* … very large, read-write region
    - contains dynamic data structures created by `malloc()` (see later)
- *stack* … dynamically-allocated data (function local vars)
    - consists of frames, one for each currently active function
    - each frame contains local variables and house-keeping info

... C execution: Memory

130/180



Exercise #16: Memory Regions

131/180

```
int numbers[] = { 40, 20, 30 };

void insertionSort(int array[], int n) {
   int i, j;
   for (i = 1; i < n; i++) {
      int element = array[i];
      for (j = i-1; j >= 0 && array[j] > element; j--)
         array[j+1] = array[j];
      array[j+1] = element;
   }
}

int main(void) {
   insertionSort(numbers, 3);
   return 0;
}
```

Which memory region are the following objects located in?

1. `insertionSort()`
2. `numbers[0]`
3. `n`
4. `array[0]`
5. `element`

---

1. code
2. global
3. stack
4. global
5. stack

Dynamic Data Structures

Dynamic Memory Allocation

134/180

So far, we have considered *static* memory allocation

- all objects completely defined at compile-time
- sizes of all objects are known to compiler

Examples:

```
int   x;      // 4 bytes containing a 32-bit integer value
char *cp;     // 8 bytes (on CSE machines)
              //   containing address of a char
typedef struct {float x; float y;} Point;
Point p;      // 8 bytes containing two 32-bit float values
char  s[20];  // array containing space for 20 1-byte chars
```

... Dynamic Memory Allocation

135/180

In many applications, fixed-size data is ok.

In many other applications, we need flexibility.

Examples:

```
char name[MAXNAME];    // how long is a name?
char item[MAXITEMS];   // how high can the stack grow?
char dictionary[MAXWORDS][MAXWORDLENGTH];
                      // how many words are there?
                      // how long is each word?
```

With fixed-size data, we need to guess sizes ("large enough").

... Dynamic Memory Allocation

136/180

Fixed-size memory allocation:

- allocate as much space as we might ever possibly need

Dynamic memory allocation:

- allocate as much space as we actually need
- determine size based on inputs

But how to do this in C?

- all data allocation methods so far are "static"
  - however, stack data (when calling a function) is created dynamically (size is known)

Dynamic Data Example

137/180

Problem:

- read integer data from standard input (keyboard)
- first number tells how many numbers follow
- rest of numbers are read into a vector
- subsequent computation uses vector (e.g. sorts it)

Example input: `6 25 -1 999 42 -16 64`

How to define the vector?

... Dynamic Data Example

138/180

Suggestion #1: allocate a large vector; use only part of it

```
#define MAXELEMS 1000

// how many elements in the vector
int numberOfElems;
scanf("%d", &numberOfElems);
assert(numberOfElems <= MAXELEMS);

// declare vector and fill with user input
int i, vector[MAXELEMS];
for (i = 0; i < numberOfElems; i++)
   scanf("%d", &vector[i]);
```

Works ok, unless too many numbers; usually wastes space.

Recall that `assert()` terminates program with standard error message if test fails.

... Dynamic Data Example

Suggestion #2: create vector after count read in

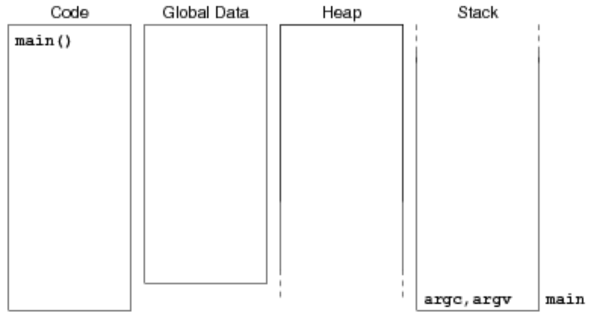
```
#include <stdlib.h>

// how many elements in the vector
int numberOfElems;
scanf("%d", &numberOfElems);

// declare vector and fill with user input
int i, *vector;
size_t numberOfBytes;
numberOfBytes = numberOfElems * sizeof(int);

vector = malloc(numberOfBytes);
assert(vector != NULL);

for (i = 0; i < numberOfElems; i++)
   scanf("%d", &vector[i]);
```

Works unless the *heap* is already full (very unlikely)

Reminder: because of pointer/array connection `&vector[i] == vector+i`

The `malloc()` function

Recall memory usage within C programs:



... The `malloc()` function

`malloc()` function interface

```
void *malloc(size_t n);
```

What the function does:

- attempts to reserve a block of `n` bytes in the *heap*
- returns the address of the start of this block
- if insufficient space left in the heap, returns `NULL`

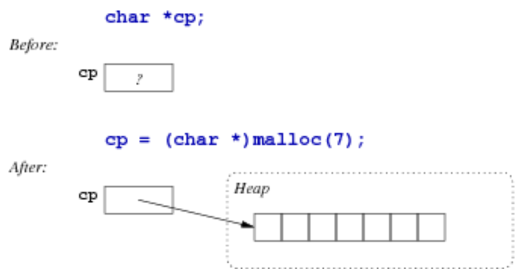
Note: `size_t` is essentially an `unsigned int`

- but has specialised interpretation of applying to memory sizes measured in bytes

---

... The `malloc()` function

Example use of `malloc`:



Note: because of `a[i] == *(a+i)` can use `cp[i]` to refer to i-th element of the dynamic array

---

... The `malloc()` function

Things to note about `void *malloc(size_t)`:

- it is defined as part of `stdlib.h`

- its parameter is a size in units of *bytes*
- its return value is a *generic* pointer `(void *)`
- the return value must *always* be *checked* (may be `NULL`)

Required size is determined by *#Elements* * `sizeof(`*ElementType*`)`

Exercise #17: Dynamic Memory Allocation

Write code to

1. create space for 1,000 speeding tickets (cf. Lecture Week 1)
2. create a dynamic $m \times n$-matrix of floating point numbers, given m and n (ensure elements can be accessed via `matrix[i][j]`)

How many bytes need to be reserved in each case?

---

1. Speeding tickets:

```
typedef struct {
        int day, month; } DateT;
typedef struct {
        int hour, minute; } TimeT;
typedef struct {
        char plate[7]; double speed; DateT d; TimeT t; } TicketT;
```

```
TicketT *tickets;
tickets = malloc(1000 * sizeof(TicketT));
assert(tickets != NULL);
```

32,000 bytes allocated

2. Matrix:

```
float **matrix;

// allocate memory for m pointers to beginning of rows
matrix = malloc(m * sizeof(float *));
assert(matrix != NULL);

// allocate memory for the elements in each row
int i;
for (i = 0; i < m; i++) {
   matrix[i] = malloc(n * sizeof(float));
   assert(matrix[i] != NULL);
}
```

$8m + 4 \cdot mn$ bytes allocated

Exercise #18: Memory Regions

Which memory region is `tickets` located in? What about `*tickets`?

---

1. `tickets` is a variable located in the stack
2. `*tickets` is in the heap (after **`malloc`**'ing memory)

... The `malloc()` function

`malloc()` returns a pointer to a data object of some kind.

Things to note about objects allocated by `malloc()`:

- they contain random values
  - need to be *initialised* before they are read
- they exist until explicitly removed (program-controlled lifetime)
- they are *accessible* while some variable references them
- if no active variable references an object, it is *garbage*

The function `free()` releases objects allocated by `malloc()`

... The `malloc()` function

The result of `malloc()` should always be *checked*:

```
int *vector, length, i;
…
vector = malloc(length*sizeof(int));
// but malloc() might fail to allocate
assert(vector != NULL);
// now we know it's safe to use vector[]
for (i = 0; i < length; i++) {
        … vector[i] …
}
```

Alternatively:

```
int *vector, length, i;
…
vector = malloc(length*sizeof(int));
// but malloc() might fail to allocate
if (vector == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
}
// now we know its safe to use vector[]
for (i = 0; i < length; i++) {
        … vector[i] …
}
```

- `fprintf(stderr, …)` outputs text to a stream called **`stderr`** (the screen, by default)
- `exit(`*v*`)` terminates the program with return value *v*

Memory Management

`void free(void *ptr)`

- releases a block of memory allocated by `malloc()`
- `*ptr` is a dynamically allocated object
- if `*ptr` was not `malloc()`'d, chaos will follow

Things to note:

- the contents of the memory block are not changed
- all pointers to the block still exist, but are not valid
- the memory may be re-used as soon as it is `free()`'d

... Memory Management

Warning! Warning! Warning! Warning!

Careless use of `malloc()` / `free()` / pointers

- can mess up the data in the heap
- so that later `malloc()` or `free()` cause run-time errors
- possibly well after the original error occurred

Such errors are very difficult to track down and debug.

Must be very careful with your use of `malloc()` / `free()` / pointers.

... Memory Management

If an uninitialised or otherwise invalid pointer is used, or an array is accessed with a negative or out-of-bounds index, one of a number of things might happen:

- program aborts immediately with a "`segmentation fault`"
- a mysterious failure much later in the execution of the program
- incorrect results, but no obvious failure
- correct results, but maybe not always, and maybe not when executed on another day, or another machine

The first is the most desirable, but cannot be relied on.

... Memory Management

Given a pointer variable:

- you can check whether its value is `NULL`
- you can (maybe) check that it is an address
- you **cannot** check whether it is a valid address
  - *dangling pointer* … points to an invalid (e.g. deallocated) memory location
  - *buffer overflow* … trying to access memory beyond allocated block

---

... Memory Management

Typical usage pattern for dynamically allocated objects:

```
// single dynamic object e.g. struct
Type *ptr = malloc(sizeof(Type));  // declare and initialise
assert(ptr != NULL);
… use object referenced by ptr e.g. ptr->name …
free(ptr);

// dynamic array with "nelems" elements
int nelems = NumberOfElements;
ElemType *arr = malloc(nelems*sizeof(ElemType));
assert(arr != NULL);
… use array referenced by arr e.g. arr[4] …
free(arr);
```

Memory Leaks

Well-behaved programs do the following:

- allocate a new object via `malloc()`
- use the object for as long as needed
- `free()` the object when no longer needed

A program which does not `free()` each object before the last reference to it is lost contains a *memory leak*.

Such programs may eventually exhaust available heapspace.

Exercise #19: Dynamic Arrays

Write a C-program that

- prompts the user to input a positive number n
- allocates memory for two n-dimensional floating point vectors **a** and **b**
- prompts the user to input $2n$ numbers to initialise these vectors
- computes and outputs the inner product of **a** and **b**
- frees the allocated memory

Sidetrack: Standard I/O Streams, Redirects

Standard file streams:

- **`stdin`** … standard input, by default: keyboard
- **`stdout`** … standard output, by default: screen
- **`stderr`** … standard error, by default: screen

- `fprintf(stdout, …)` has the same effect as `printf(…)`
- `fprintf(stderr, …)` often used to print error messages

Executing a C program causes `main(…)` to be invoked

- with `stdin`, `stdout`, `stderr` already open for use

... Sidetrack: Standard I/O Streams, Redirects

158/180

The streams `stdin`, `stdout`, `stderr` can be *redirected*

- redirecting `stdin`

```
prompt$ myprog < input.data
```

- redirecting `stdout`

```
prompt$ myprog > output.data
```

- redirecting `stderr`

```
prompt$ myprog 2> error.data
```

Linked Lists as Dynamic Data Structure

Sidetrack: Defining Structures

160/180

Structures can be defined in two different styles:

```
typedef struct { int day, month, year; } DateT;
// which would be used as
DateT somedate;

// or

struct date { int day, month, year; };
// which would be used as
struct date anotherdate;
```

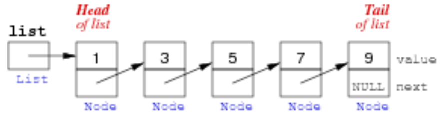
The definitions produce objects with identical structures.

It is possible to combine both styles:

```
typedef struct date { int day, month, year; } DateT;
// which could be used as
DateT       date1, *dateptr1;
struct date date2, *dateptr2;
```

Self-referential Structures

161/180



Reminder: To realise a "chain of elements", need a *node* containing

- a value
- a link to the next node

In C, we can define such nodes as:

```
typedef struct node {
   int data;
   struct node *next;
} NodeT;
```

... Self-referential Structures

162/180

Note that the following definition does not work:

```
typedef struct {
   int data;
   NodeT *next;
} NodeT;
```

Because `NodeT` is not yet known (to the compiler) when we try to use it to define the type of the `next` field.

The following is also illegal in C:

```
struct node {
   int data;
   struct node recursive;
};
```

Because the size of the structure would have to satisfy `sizeof(struct node) = sizeof(int) + sizeof(struct node) = ∞`.

Memory Storage for Linked Lists

163/180

Linked list nodes are typically located in the heap

- because nodes are dynamically created

Variables containing pointers to list nodes

- are likely to be local variables (in the stack)

Pointers to the start of lists are often

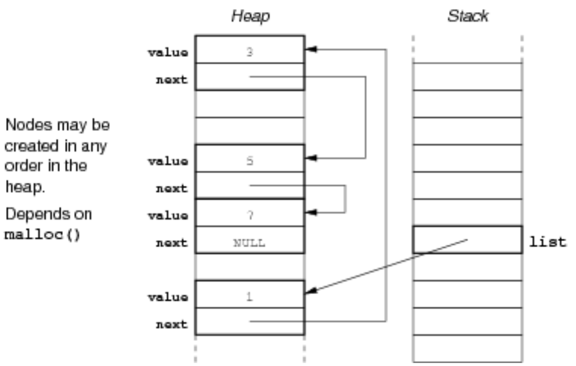
- passed as parameters to function
- returned as function results

... Memory Storage for Linked Lists

Create a new list node:

```
NodeT *makeNode(int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;        // initialise data
   new->next = NULL;     // initialise link to next node
   return new;           // return pointer to new node
}
```

... Memory Storage for Linked Lists



Iteration over Linked Lists

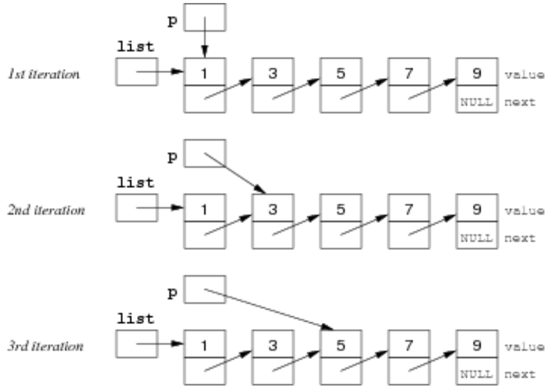
When manipulating list elements

- typically have pointer `p` to current node (`NodeT *p`)
- to access the data in current node: `p->data`
- to get pointer to next node: `p->next`

To iterate over a linked list:

- set `p` to point at first node (head)
- examine node pointed to by `p`
- change `p` to point to next node
- stop when `p` reaches end of list (`NULL`)

... Iteration over Linked Lists



... Iteration over Linked Lists

Standard method for scanning all elements in a linked list:

```
NodeT *list;  // pointer to first Node in list
NodeT *p;     // pointer to "current" Node in list

p = list;
while (p != NULL) {
        … do something with p->data …
        p = p->next;
}

// which is frequently written as

for (p = list; p != NULL; p = p->next) {
        … do something with p->data …
}
```

... Iteration over Linked Lists

Check if list contains an element:

```
int inLL(NodeT *list, int d) {
   NodeT *p;
   for (p = list; p != NULL; p = p->next)
      if (p->data == d)       // element found
         return true;
   return false;              // element not in list
}
```

Print all elements:

```
void showLL(NodeT *list) {
   NodeT *p;
   for (p = list; p != NULL; p = p->next)
```

```
    printf("%6d", p->data);
}
```

Modifying a Linked List

170/180

Insert a new element at the beginning:

```
NodeT *insertLL(NodeT *list, int d) {
   NodeT *new = makeNode(d);  // create new list element
   new->next = list;          // link to beginning of list
   return new;                // new element is new head
}
```

Delete the first element:

```
NodeT *deleteHead(NodeT *list) {
   assert(list != NULL);  // ensure list is not empty
   NodeT *head = list;    // remember address of first element
   list = list->next;     // move to second element
   free(head);
   return list;           // return pointer to second element
}
```

What would happen if we didn't `free` the memory pointed to by `head`?

Exercise #20: Freeing a list

171/180

Write a C-function to destroy an entire list.

Iterative version:

```
void freeLL(NodeT *list) {
   NodeT *p, *temp;

   p = list;
   while (p != NULL) {
      temp = p->next;
      free(p);
      p = temp;
   }
}
```

Why do we need the extra variable `temp`?

Abstract Data Structures: ADTs

Abstract Data Types

174/180

Reminder: *Abstraction* …

- is an approach to implementing data types
- separates *interface* from *implementation*
- users of an ADT see only the interface
- builders of an ADT provide an implementation

E.g. does a client want/need to know how a Stack is implemented?

... Abstract Data Types

175/180

We want to distinguish …

- ADO = *a*bstract *d*ata *o*bject (e.g. a *single* stack, as in week 1)
- ADT = *a*bstract *d*ata *t*ype (allows users to create their own data objects)

Warning: Sedgewick's first few examples are ADOs, not ADTs.

Typical operations with ADTs

- *create* a value of the type
- *modify* one variable of the type
- *combine* two values of the type

... Abstract Data Types

176/180

ADT *interface* provides

- an *opaque* user-view of the data structure (e.g. `stack *`)
- function signatures (prototypes) for all operations
- semantics of operations (via documentation)

ADT *implementation* gives

- concrete definition of the data structure
- function implementations for all operations
- … including for *creation* and *destruction* of instances of the data structure

ADTs are important because …

- facilitate decomposition of complex programs
- make implementation changes invisible to clients
- improve readability and structuring of software

Stack as ADT

177/180

Interface (in `Stack.h`)

```
// provides an opaque view of ADT
typedef struct StackRep *stack;
```

```
// set up empty stack
stack newStack();
// remove unwanted stack
void dropStack(stack);
// check whether stack is empty
bool StackIsEmpty(stack);
// insert an int on top of stack
void StackPush(stack, int);
// remove int from top of stack
int StackPop(stack);
```

ADT `stack` defined as a *pointer* to an *unspecified* struct named `StackRep`

Stack ADT Implementation

Linked list implementation (`Stack.c`):

Remember: `Stack.h` includes `typedef struct StackRep *stack;`

```
#include <stdlib.h>
#include <assert.h>
#include "Stack.h"

typedef struct node {
   int data;
   struct node *next;
} NodeT;

typedef struct StackRep {
   int    height;   // #elements on stack
   NodeT *top;      // ptr to first element
} StackRep;

// set up empty stack
stack newStack() {
   stack S = malloc(sizeof(StackRep));
   assert(S != NULL);
   S->height = 0;
   S->top = NULL;
   return S;
}

// remove unwanted stack
void dropStack(stack S) {
   NodeT *curr = S->top;
   while (curr != NULL) {  // free the list
      NodeT *temp = curr->next;
      free(curr);
      curr = temp;
   }
   free(S);           // free the stack rep
}
```

```
// check whether stack is empty
bool StackIsEmpty(stack S) {
   return (S->height == 0);
}

// insert an int on top of stack
void StackPush(stack S, int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;
   // insert new element at top
   new->next = S->top;
   S->top = new;
   S->height++;
}

// remove int from top of stack
int StackPop(stack S) {
   assert(S->height > 0);
   NodeT *head = S->top;
   // second list element becomes new top
   S->top = S->top->next;
   S->height--;
   // read data off first element, then free
   int d = head->data;
   free(head);
   return d;
}
```

Common Mistakes

Warning! Warning! Warning! Warning!

- Never `#include` an ADT *implementation* file

```
#include "Stack.c"     // correct is: #include "Stack.h"
```

- Do not try to access struct details in a client

```
#include "Stack.h"
...
stack S = newStack();
...
if (S->height == 0) {   // correct is: StackIsEmpty(S)
...
```

Summary

- Big-Oh notation
- Asymptotic analysis of algorithms
- Examples of algorithms with logarithmic, linear, polynomial, exponential complexity
- Linked lists vs. arrays
- Pointers
- Memory management
  - **`malloc()`**
    - aim: allocate some memory for a data object
      - the location of the memory block within heap is random
      - the initial contents of the memory block are random
    - if successful, returns a pointer to the start of the block
    - if insufficient space in heap, returns `NULL`
  - **`free()`**
    - releases a block of memory allocated by `malloc()`
    - argument must be the address of a previously dynamically allocated object
- Dynamic data structures

- Suggested reading:
  - big-Oh notation … Sedgewick, Ch. 2.1-2.4, 2.6
  - pointers … Moffat, Ch. 6.6-6.7
  - dynamic structures … Moffat, Ch. 10.1-10.2
  - linked lists, stacks, queues … Sedgewick, Ch. 3.3-3.5, 4.4, 4.6

Produced: 13 Jan 2025