2/21

Recurrence Relations with Full History: By the end of Today you should know everything for ~~section~~ chap. 3

simplest form:

$$T(n) = c + \sum_{i=1}^{n-1} T(i).$$

Then $T(n+1) - T(n) = T(n)$. Thus $T(n+1) = 2T(n)$ which implies $T(n+1) = 2^n T(1)$ (also correct for $T(1)$)

But say if $T(1) = 1$ and $c = 5$ $T(2) = 6 \neq 2T(1)$ The base case is $T(2) - T(1) = c \neq T(1)$

Thus $T(2) = T(1) + c$ (by definition) and $T(n+1) = 2T(n)$ for $n > 2$. Hence $T(n+1) = (T(1)+c)2^{n-1}$

Note that c didnot appear in the formula which was strange. Solution: always try for more base cases to avoid these cases.

The next one is not so simple useful for Quicksort that we discuss soon.

$$T(n) = n-1 + \frac{2}{n}\sum_{i=1}^{n-1} T(i), \text{ for } n \geq 2,\ T(1) = 0 \Rightarrow n\,T(n) = n(n-1) + 2\sum_{i=1}^{n-1} T(i) \quad (*)$$

$$T(n+1) = (n+1) - 1 + \frac{2}{n+1}\sum_{i=1}^{n} T(i) \qquad \Rightarrow (n+1)T(n+1) = (n+1)n + 2\sum_{i=1}^{n} T(i) \quad (**)$$

$$(**) - (*) = (n+1)T(n+1) - n\,T(n) = (n+1)n - n(n-1) + 2T(n) \Rightarrow T(n+1) = \frac{n+2}{n+1}T(n) + \frac{2n}{n+1} \text{ for } n \geq 2$$

To get a close approximation replace $\frac{2n}{n+1}$ by 2.

$$T(n) \leq 2 + \frac{n+2}{n+1}T(n) \leq 2 + \frac{n+2}{n+1}\left[2 + \frac{n+1}{n}\left[2 + \frac{n}{n-1}\left(2 + \frac{n-1}{n-2}\left(\cdots 1\right)\right)\right)\right. \quad T(2) = 1$$

$$= 2\left[1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \frac{n+2}{n+1}\frac{n+1}{n}\frac{n}{n-1} + \cdots - \quad + \frac{n+2}{n+1}\frac{n+1}{n}\frac{n}{n-1}\cdots 1\right]$$

$$= 2\left[1 + \frac{n+2}{n+1} + \frac{n+2}{n} + \frac{n+2}{n-1} + \cdots \quad \frac{n+2}{1}\right] = 2(n+2)\left[\frac{1}{n+2} + \frac{1}{n+1} + \frac{1}{n} + \cdots + 1\right] = 2(n+2)\,H(n)$$

where $\underline{H(n)} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n} = \ln(n) + \gamma + O(\frac{1}{n})$ where $\gamma = 0.577\ldots$ Thus $H(n) = O(\log n)$

Harmonic series

and $T(n) \leq 2(n+2)H(n)$ and $T(n) = O(n \log n)$

Useful formulas: Arithmetic series $1 + 2 + \cdots n = \frac{n(n+1)}{2}$ if $a_n = a_{n-1} + c$, $a_1 + a_2 + \cdots a_n = \frac{n(a_n + a_1)}{2}$ c is constant

Geometric series: $1 + 2 + \cdots + 2^n = 2^{n+1} - 1$ if $a_n = c\,a_{n-1}$ $\Rightarrow$ $a_1 + a_2 + \cdots a_n = a_1\frac{c^{n-1}-1}{c-1}$ $c \neq 1$ a constant [if $0 < c < 1$ then $\sum_{i=1}^{\infty} a_i = \frac{a_1}{1-c}$

sum of squares: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ Harmonic series $H_n = \sum_{k=1}^{n}\frac{1}{k} = \ln n + \gamma + O(\frac{1}{n})$ wher $\gamma = 0.577$ is Euler's constant

$\log_b a = \frac{1}{\log_a b}$, $\log_a x = \frac{\log_b x}{\log_b a}$, $b^{\log_b x} = x$, $b^{\log_a x} = x^{\log_a b}$

Sum of logarithms: $\sum_{i=1}^{n}\lfloor \log_2 i \rfloor = (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2 = \Theta(n \log n)$

Summation by integral: $\sum_{i=1}^{n} f(i) \leq \int_{x=1}^{x=n+1} f(x)\,dx$ for f monotone increasing continuous function

Stirling's approximation
$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + O(\frac{1}{n})\right)$
Thus $\log_2(n!) = \Theta(n \log n)$

Feb 21, 2011

A Brief Intro to Data structurs:

Data structures are the building blocks of computer Algorithms. Here we consider abstract data type(ADT) (e.g. do not specify the data types like integers, reals, characters). It is more convenient and more general to design the algorithms for these operations without specifying the data type of the items.

Modern object-oriented languages such as C++ or Java support a form of abstract data. When a class is used as a type, it is an abstract type that refers to hidden representation. An ADT is typically implemented as a class and each instance of the ADT is an object of that class.

Elementary Data structures:

Elements: It is a generic name for an unspecified data type. An element can be an integer, a set of integers, a file of text, or another data structure. we use this term whenever the discussion is independent of the type of data. For example in sorting we only care about comparisons between elements which can be integers, names (strings of characters), etc. Usually we assume we can compare elements for equality (and some time "less than") and we can copy them. And each of these operations are counted one unit of time.

Array: An array is a row of elements of the same type. The size of an array is the number of elements in that array. The size of the array should be fixed and allocated in memory in advance. The elements can be bytes, integers, strings or more generally any elements.

Arrays are very efficient and very common data structures and each element of it can be accessed in constant time. Arrays are good except they store elements of the same type and their size is fixed. We represent them as $a[1..n]$ and acess them with $a[i], 1 \leq i \leq n$ or we can have two dimensions $a[1..n, 1..m]$ and access them with $a[i,j]$.

Records: Records are similar to arrays, except that we do not assume all elements are of the same type. A record thus a list of elements of different types. However like arrays the exact combination of types and thus the storage size is known. Each element in a record can be accessed in constant time. (since the sizes of all the elements in the records are known it is possible to compute the location of each element in constant time. Example:

```
record example1
begin
   Int 1: integer;
   Ar1: array [1..20] of integers;
  Name 1, array [1..12] of characters;
end.
```

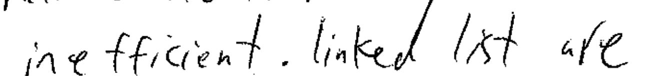
~~classes~~ ~~for~~ structures in C and classes in C++ (indeed more general) can be used to implement records.

linked lists:

There are many applications in which the number of elements is changing dynamically as the algorithm progresses (we can have large arrays to satisfy them, but it is often not a good solution. Also due to consecutive representation of arrays doing some operations like insertion and deletions in the middle of an array is very inefficient. linked list are simplest form of dynamic data structures. Here each element is represented separately and all elements are connected through the use of pointers.

→A pointer is simply a variable that holds as its value the address of another element and it exists in C, C++, but not explicity in Java (but can be simulated)

→A linked list is a list of pairs, say in a record, each containing an element and a pointer, such that each pointer contains the address of the next pair. It is not possible to access each element directly in a linked list, instead you need to scan it by following the addresses in the pointers called a linear scan. Example



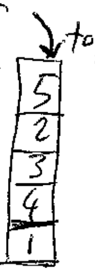
Two drawbacks { 1- requirs more space (one additional pointer per element) [Not a huge problem these

2- we need to do linear scan to find say 30th element.

Advantage { with only two operation we can insert an element in the middle. and with one chang we can delete an element.

The end of the list will be denoted by nil or null, a pointer ~~which~~ a pointer to nowhere

Stacks: a dynamic data structure in which we delete (Pop) an element most recently inserted (push). A stack implements a last-in first-out (LIFO) policy. we have also an operation top which gives the element at the top of the stack. we can implement it with both arrays (if we know the maximum size) (see below) or the linked list (otherwise)

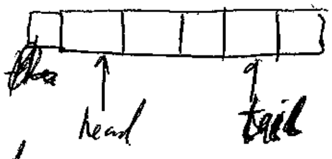


```
TOP(S)
    if t = 0 the return -1
    else return S[t]

push(S, x)
    t ← t+1
    S[t] ← x

Pop(S)
    if t = 0 then error "underflow"
    else t ← t-1
         return S[t+1]
```

Each of push, Pop and top takes O(1) time.
As an exercise you can implement it with linked lists.

Queues: a dynamic data structure in which the element deleted (Dequeue) is always the one that has been inserted (Enqueue) for the longest time. the queue implements a first-in first-out (FIFO) policy like in the registerar's office. The queue has a head and a tail. (insert at the tail, delete from the head). Again it can be implemented by array (see below) or linked lists (like stacks). By array we implement it as a circular way.

```
Enqueue(Q, x)
    Q[tail] ← x
    if tail = lengthQueue
        then tail ← 1
        else tail ← tail+1

Dequeue(Q)
    x ← Q[head]
    if head = lengthQueue
        then head ← 1
        else head ← head+1
    return x.
```



Again each operation takes O(1) time. Implement it by linked list as an exercise.