LectureNotes

PIC 10B

PROFESSOR POLLETT

FALL 2000

LectureNotes

Fall 2000

PROGRAM IN COMPUTING 10B
PROFESSOR POLLETT
SET #1

SEPTEMBER 29, 2000

Web page for class : http://www.math.ucla.edu/~cpollett
Prof. Pollet's OH: MWF 10-11am

1st Hw is already on the web. Due next Friday (Oct.6) for lectures 1 and 2.

Plan for today (9/29/00, Friday): Go over syllabus and talk about classes in C++.

Major difference between 10A and 10B
In 10B we will talk about abstract data types which are useful in any language: lists, list processing, various of kind of trees, sorting algorithms, hashing. We'll also be concerned with the efficiency of our algorithms. So we'll learn about tools to measure this such as O-notation.

Classes in C++ (Ch. 6.2 in Savitch)
A class is a C++ construct used to group several related variables. and functions together (this grouping is called encapsulation).

Ex 1 ostream, ofstream, istream, ifstream are classes for input and output. An example object of type ostream is cout. cout has some member functions such as cout.setprecision(3);

How do define a class

```
class EmployeeAcct          //Name of class: notice that 1st letter of each word is capitalized.
{public:
    EmployeeAcct();             //Constructors say how to make this kind of object.
    EmployeeAcct(int amt);
    int getAmt();
private:
    int amount;             //Where data is stored.
};
```

To create an object of type EmployeeAcct in main or some other function we could do:

```
int main ()
{   EmployeeAcct a, b(10);      // a is created using 1st constructor:  a( ) is not legal C++. b(10) is created using 2nd constructor.
// to access info stored in a and b
    cout << a. getAmt()<<endl;    //prints 0
    cout<< b. getAmt()<<endl;     //prints 10
}
```

Dot operator is used to access member variables or functions.

END OF LECTURE ***

OCTOBER 2, 2000

Reminder: HW 1 is due Friday.
Last lecture: talked about classes in C++
Today: We will talk about 1) How to define member functions
2) Difference between public and private
3) ADT's

```
class EmployeeAcct
{
    public:
        EmployeeAcct( );
        EmployeeAcct(int amt);
        int getAmt( );
    private:
        int amount;
};
```

To define member functions:

```
EmployeeAcct::EmployeeAcct( )       // This would appear outside }; ending the class.
{    amount = 0;    }

                                    //  Two colons are called scope resolution operator.

                                    //  which class we are defining member function of

EmployeeAcct::EmployeeAcct(int amt)
{    amount = amt;     }

int EmployeeAcct::getAmt( )                 // int is the return type
{    return amount;     }
```

Now the class has been defined. An example of how to use the class:

```
int main ( )
{    EmployeeAcct a, b(10);          //  a uses the 1st constructor
     cout << a.getAmt( ) << endl;    // b uses the 2nd constructor
     cout << b.getAmt( ) << endl;
     return 0;
}
on screen:    0
              10
```

Public vs. Private

What's the difference?

Public variables/functions can be accessed by any other object or function.

(Ex) In main above we accessed a's getAmt function

In contrast...

Private variables can only be accessed within the scope of that class

(Ex) Above could access amount when defining EmployeeAcct constructors since within the scope of EmployeeAcct class. It would be a criminal offense to do this in main since not in scope of EmployeeAcct class

Why bother having the distinction?

Allows us to hide the way we're storing the data internally in case we need to change it. This is called Information Hiding.

More of this in a sec...

First some definitions:

A data type consists of a collection of values together with a set of base operations defined on these values.

(Ex) Data type int has ==, +, -, *, /...

A data type is an abstract data type (ADT) if the programmer who uses the type does not have access to the details of how the values are implemented.

(Ex) For data type int you do not know how code for + is actually written.

Classes allow us to define ADT's: what J.Programmer sees is public part.

Private part is how we actually do the implementation.

(Ex) class Date

```
    {    public:
             Date(int day, int month, int yr, int calenderType);          // For calendarType 0-Gregorian  1-Julian  etc....
             Date(int day, int month, int yr);       // calenderType is assumed to be Gregorian.
             int getDay(int calType);
             int getMonth(int calType);
             int getYear(int calType);
             int addDay( );
             void output(int calType);
         private:
    // When we write this we have to make decisions about what is stored internally.
    // We assume information stored in Gregorian format.
```

```
        int d;
        int m;          // Or internally we could have stored just the total number of days since year zero.
        int y;

  };
```

Public part is called the interface. Private part is called the implementation.
The point is as long as we write our public member functions correctly outsiders never need to know how we stored things.

```
Date::Date( )
{    d=0;
     m=0;
     y=0;
}
        // could define other member functions.
```

Above example does not show how to define ==, +, *, etc. for an ADT
suppose we had

```
    Date d1, d2(1, 1, 1), d3(1, 1, 2);
    // then
    d1 = d2;  // statement is legal and it sets date inside d1 to be same as d2
              // Every private member of d1 set to corresponding value in d2.
    // if we did
    if (d1 == d2) cout << "hi there" << endl;
    // statement is not legal at this point since we have not defined == operator
    // for Date class
```

**END OF LECTURE **

OCTOBER 4, 2000

Reminder: HW1 due Friday. Make sure to get file name right.

Last Day - talked about defining member functions for classes. Talked a little about ADT's. Mentioned operator overloading.
Today - mainly talk about operator overloading. At the end of today we will start talking about separate compilation.

Last day - we talked about class Date. We said we couldn't do

```
Date d1, d2;
if (d1 == d2) cout << "hi\n";
```

yet since == wasn't defined for this class.

To define == for Date ...

```
class Date
{ public:
     friend bool operator ==(const Date& d1, const Date& d2);
     : // rest of class
};
```

friend - this keyword means that the function given after it is not a member function of the class, but is allowed access to any private data stored in objects of this class.

```
const  Date& d1 // & means call-by-reference i.e., get address at where a Date object is stored
  ↑
//means we will not change
```

Now outside of above class definition we'd write ...

```
bool operator ==(const Date& d1, const Date& d2)
{  return ( d1.d == d2.d &&
            d1.m == d2.m &&
            d1.y  == d2.y);
}
```

allowed access to these private numbers variables since this was a friend function of Date.

```
class Hour  //class to store roughly the hour hand from a clock. Stores 0 to 11.
{ public:
     Hour( );
     Hour(int h);
     int getHour( );
     friend bool operator ==(const Hour& h1, const Hour& h2);
     friend Hour operator +(const Hour& h1, const Hour& h2);
     //could do -, /, %, * this way.
     friend ostream& operator <<(ostream& out, const Hour& h);
     friend istream& operator  >>(istream& in, const Hour& h);
private:
     int hour;
};
// Let's define member functions
Hour::Hour( )
{    hour = 0;       }
Hour::Hour(int h)
{    hour = h%12;       }    // If entered 23 then output 11th hour.
int Hour::getHour( )
{    return hour;       }
bool operator ==(const Hour& h1, const Hour& h2)
{    return h.hour == h2.hour);
}
Hour operator +(const Hour& h1, const Hour& h2)
{    return  ((h1.hour+h2.hour)%12);
}
//before defining << and >> let me explain a little.
```

When we have an expression like
cout << "hi there" << "you";
It's like we'd written (cout << "hi there) << "you";

```
                          ↑
                        prints hi there to the screen and returns the cout object.
```

Then (cout << "you"); is evaluated.

```
          ↑
prints you to screen and outputs cout which since no more << operators does nothing.
```

This kind of evaluation explains prototype for << object.

```
ostream& operator <<(ostream& out, Hour& h)
      ↑                    ↑               ↖
usually address of cout    usually cout       thing to be printed
```

```
ostream& operators <<(ostream& h)
{ out << "Hour:"
     <<h.hour<<endl;
  return out;
}
istream& operator >>(istream& in, const Hour& h)
{ int hour;
  in >> hour;
  h.hour = hour%12;
  return in;
}
```

END OF LECTURE **

OCTOBER 6, 2000

Last day - talked about operator overloading
Today - separate compilation of files and dynamic allocation of objects and arrays. If have time we will talk about the this pointer.

Separate Compilation
It is often the case that many people work on the same project. We would like to be able to split large programs into several files so that people working on a project can work independently.
Standard way to do this . . .
Usually split a program based on class definitions.

Ex: Consider the Hour class we talked about last time. We'd put the interface in a file Hour.h and the implementation into a file Hour.cpp.

Hour.h

```
class Hour
{ Hour( );
     . . . .
};
```

Hour.cpp

```
#include
     "Hour.h"
Hour::Hour()
{    // code
}
"other member functions
```

→ preprocessor directive to prepend Hour.h to this file

Anyone who wants to use the Hour class tells the compiler by putting #include "Hour.h" at the start of their file.

Someones.cpp

```
#include
     "Hour.h"
// their code
```

Why do we separate the code for the interface and implementation into two files?
Idea is that the user of class does not need to know the details of how class is implemented. User just needs to know member functions and what they're supposed to do. Also looking through a file with only the interface is easier to do than looking through both interface and implementation.
One problem with this setup . . .

User1.h

```
#include
     "Hour.h"
```

User2.h

```
#include
     "Hour.h"
#include
     "User1.h"
```

// Could end with two copies of Hour.h in front of file that gets compiled.
// Causes an error.

We'd like Hour.h to be smart enough to "know" if it's already been loaded.
To make Hour.h smart use #ifndef

Hour.h

```
#ifndef HOUR_H   //preprocessor flag
#define HOUR.H   // here says flag has been defined
class Hour
{    // our code
};
#endif    // end #ifndef HOUR.H
```

Dynamically Allocating Arrays and Objects
Why do we want to do this? Sometimes we would like to set size of an array at runtime or create an object at runtime.
To do this for objects. . . .

```
Ex. Hour *h;       // creates a pointer to an object of type Hour, i.e., h can
                   // store the memory address of where something of type
                   // Hour lives.
    h = new Hour();    // creates at runtime an object of type Hour using default
                       // constructor
    //To use member functions of this object could do things like:
    cout<<h→getHour()<<endl; // prints 0 to the screen
         ↑
      //Same as (*h).getHour();
      //(*h) means object stored at address h.

    //When we're done using this object we free up memory by doing . . .
    delete h; // frees memory used by h.   If h has a destructor call it.
```

To do this for arrays . . .

Ex
```
    int *myarr;
    int size;
    cin >> size;
    myarr = new int[size];  // Creates an array of size many ints, returns its
                            // location to myarr.
    //To set a value of this array
    myarr[4] = 6; // Sets 5th element since we start counting at 0.
     ↑
    //this is equivalent to
    *(myarr + 4) = 6;  //move over 4 ints from n and store 6.
    //To get rid of this array when done . . .
    delete[ ] myarr;     //says we're deleting an array
```

Like to start writing class to illustrate what we've talked about.

```
#ifndef ARRAY_H
#define ARRAY_H
// class for dynamic arrays
class Array
{ public:
     Array(int s);  // allows us to create an array of size s.
     Array( );      // default array size is 10.
     Array(const Array& arr);    // copies arr into current object.  Called
                                 // a copy constructor
     ~ Array( );    // destructor frees up memory used by object
     Array& operator =(const Array& rhs) // overloads = so that copies dynamically allocated
                                         // array properly.  Notice this is a member function
                                         // not a friend function
     // other methods
     private:
          int *array;    // where the data is stored
          int max;       // how big array is.
};
#endif
```

END OF LECTURE AND SET #1 ***

LectureNotes *Fall 2000*

PROGRAM IN COMPUTING 10B
PROFESSOR POLLETT
SET #2

OCTOBER 9, 2000

Reminders: HW 2 is on the web. Section 2C now meets Bunche 3150. HW 1 solutions are posted.
Last day-Separate compilation. Started on example of separate compilation that also was going to give an example of the pointer.

```
#ifndef ARRAY_H
#define ARRAY_H
class Array
{
            // stuff
};
#endif
```

Today-We will implement Array class. Talk about = overloading and the this pointer. We will also begin doing algorithm efficiency analysis.

```
//To implement Array make a tile called Array.cpp
#include<iostream.h>
#include<stdlib.h>     //for exit( )
#include<stddef.h>     //for NULL constant
Array::Array( )
{    max=100;              //default array size
     array=new int[max];
     if (array==NULL)                          //ascertain if memory really got allocated
     {    cout "Error: not enough memory";
          exit (1);
     }
}
Array::Array(int s)
{    'max=s;
     array=new int[max];
     if (array== NULL)
     {    cout << "Error: not enough memory";
          exit(1);
     }
}
Array::size( )
{    return max;   }
Array::Array(const Array& arr)   //copy constructor: argument is an array we are going to copy
{    max=arr.size( );
     array=new int[max];
     if (arry == NULL)   { //same code as before }
     for (int i=0; i< max; i++)     array[i] = arr.arry[i];
}
Array::~Array( )   //destructor is called when we delete an array.
{    delete[ ] array;     }
Array& Array::operator =(const Array& rhs)
{    //this is a member function not a friend
     if (this == &rhs)   //"this" pointer is always a pointer to the current object. Left hand size of the operator is what this refers to.
     {return *this;}          //So in a=b this refers to object a.
// equals has a return value. That way, expressions like a=(b=c) makes sense. Also if (a=b) means set a equal to b then use the value of a in if
// condition.
// now we handle a case where two sides of equality not the same
     else
     {    delete[ ] array;
          max = rhs.size( );
          array = new int[max];
          if (array == NULL)       {     //same code as before    }
          for (int i=0; i<max; i++) array [i]=rhs.array[i];
     }
}
//other member functions code
```

Then in a new file myFile.cpp we could use above class.

```
#include <iostream.h>
#include "Array.h"
int main( )
{    Array a(10), b; c(a);
                  ↑
//uses default constructor         // uses copy constructor
c=a; ...  etc.
// uses overloaded =. c has same value as a but not same address, thus else clause is evaluated because conditional expression of if clause is
//false.
}
```

Can also do other oprator overloading with member functions.

```
ex.  class A
     { A& operator +(A& b);      // left hand side is passed using this.
}
     int main ( )
     { A c,d;
          cout << c+d;
}
// There is a reason you might not want to do the above for +, but use friend function instead: Often times you would like to say what adding an
//integer constant to class means.
//for example, suppose you have a class RealNumber. Then if a is a real number, a+10 makes sense. But if overload + as a member function
// a+10 will make sense whereas 10+a won't.
```

END OF LECTURE ***

OCTOBER 11, 2000

Made some corrections to HW2 yesterday. Prof. Pollett notified students by email. Send Prof. Pollett on email if you didn't receive one from him.

Last Day - talked about the this pointer and overloading + operator
Today - mention some things about copy constructors and start talking about now to analyze algorithms.

Copy Constructors

copy constructors can be explicitly called as in...
Array a, c(a); ← //copy constructor explicitly called for object c.
On the other hand sometimes copy constructors called automatically.

```
Ex. Void MyFunction(Array b)    ← //call by value
     {    //some code
     }
     int main ( )
     {    Array a(10)
          MyFunction(a);
          return 0;
     }
//when b is created the copy constructor is used to copy a's value into b.
//if you don't have a copy constructor then b is given the same reference as a. This can cause problems as the destructor b called when
//MyFunction. So also a's will be called as same.
//So important to have a copy constructor!
```

Here's another situation where this comes up.

```
Array Function2( )
{   Array b(10);
    return b; ← //just before b gets deleted (since it's a local variable)
}                   //copy constructor called and the object created is what comes back from return.
```

Analyzing algorithms

ex. consider a situation similar to HW2. We want to have a dynamically sized array and have an operation + which allows us to add one integer to what's stored in the array.

ex. initial array

index = 0

// last place where we've stored stuff

+5 // now if we add 5 to array we change our dynamic array to

5

index = 1

//if we want to now add 6 to this array our ADT will handle this by dynamically creating a new larger array by copying the contents of our old

//array into new array, adding 6 to the new array and deleting old array.

//How big should the new array be?

//Let's say we just increased its size by 1, then

5	6

index = 2 // we've copied 1 element to make this array from old array.

// if now we add 3 we would get

5	6	3

index = 3 // we've copied 2 elements in going to this array from previous array.

// generalizing, if we add n elements then how many copies do we make totally?

$$\# \text{ of copies} = 1+2+3+\ldots+(n-2)+(n-1) = \frac{(n-1)n}{2} \approx n^2$$

//roughly, n^2ish many copies.

//Since # of copies determines the speed of the algorithm, we ask if we can do better?

//Answer: yes, we can do better if when we resize array we create an array twice as large as the old array.

ex.

1

index = 1

↓ + 5

1	5

//if we now add 6

1	5	6	

index=3 //now only resize if index=size of array

+3

1	5	6	3

index=4 //no copies were made in this step.

//If we add in 2

1	5	6	3	2			

index=5

//so if we add n elements how many copies will we make?

//first how many times did we increase array size?

//Biggest the array gets is at most twice the number of elements inserted

//Array size is a power of 2. So number of times we double is the least x such that $2^x > n$

//so 2^x=n if x=$\log_2$n. So 2^x is greater than n where x=[$\log_2$n]

//so the number of copies we do with this scheme is bounded by

//$1+2+2^2+\ldots+2^{[\log_2 n]-1}$

//$=(1+2+2^2+\ldots+2^{[\log_2 n]-1})(2-1)$

//$=(2+2^2+\ldots+2^{[\log_2 n]-1}+2^{[\log_2 n]})-(-1-2-2^2-\ldots-2^{[\log_2 n]-1})$

//$=2^{[\log_2 n]}-1 \approx n$

//roughly n which is smaller than n^2ish (the speed of the first algorithm)

//so second algorithm is better

//second algorithm is used in vector class in STL (standard library)

END OF LECTURE **

OCTOBER 13, 2000

Added a bonus problem to HW2. Bonuses are worth 1 point each and are added on top of your curved final grade.

Last day - We analyzed some algorithms for dynamically sized arrays.
Today - We will talk more about analyzing algorithms and we will talk about growth rates of various computer science functions.

How to figure how good is an algorithm.
One way...
Empirical analysis
Figure out algorithm behavior by running a program. Implementing it on various kinds of inputs (like software testing).

Three types of test data
1. Actual data - data similar to what will occur for your applications of the algorithm.
2. Random data - randomly generating inputs to your algorithm.
3. Perverse or adverserial data - data designed to make algorithm as inefficient as possible.
Problem with empirical analysis is that it can be very machine and operating system dependent.

Another approach is...
Comparative analysis
Compare different algorithms for same problem based on the number of times certain they use certain base operations. Ex. copying one element from an old array to a new one. As with Empirical Analysis, we can compare algorithms on worse-case or adverserial inputs or on average-case inputs. So to do this kind of comparative analysis, we need to understand how to compare growth rates of various functions of input.

Typical growth rates one sees in Computer Science
1 - cost of one base operation (constant time).

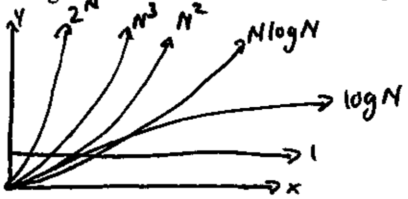
log N - (log in computer science is always base 2 since we're dealing with binary numbers). log N is roughly the length of N written in binary.
log 2 = 1, log 1 = 0, log 5 > 2 (since log 4 = 2)
Ex. Search for a number less than N using only questions of form, "Is number bigger than x?"
Number of steps to find number x log N.

N - linear growth
Ex. Scanning a file of length N takes N reads

N log N - it had to do binary search as in log N example N times.
N^2 - two rested for loops of same size
N^3 - three rested for loops of same size.

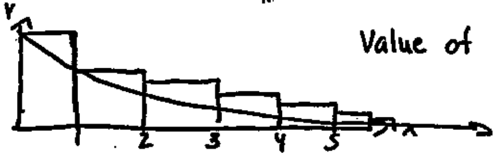
2^N - look at all possible strings of length N and do something with them.

Drawing a picture of how these functions look like:



Fractions don't exist in computer science world.
Ex. [x] - round x down
[2.5] = 2
[x] - round x up
[2.5] = 3
Ex. $[\log_2^{(x+1)}] = |x|$ (length of x)

Estimating function growth rates with using integrals.
The Harmonic Series $H_N = 1 + 1/2 + 1/3 + 1/4 + ... + 1/N$



Value of Hn is the sum of the areas in the boxes.

So lower band is given by $\int_1^N \frac{1}{x}dx = \log N$

In fact, $H_N \approx \ln N + \gamma + \frac{1}{12N}$ Where γ is Euler's gamma constant approximately $0.5d_1d_2...$

Ex. 2: Consider function N! = N(N-1) ... 1

To get a lower bound note:

$$\ln N! = \sum_{i=1}^{N} \ln i \geq \int_1^N \ln x dx = N \ln N - N$$

Actually, Stirling's Formula says $\ln N! \approx N\ln N - N + \ln\sqrt{2\pi n}$

so $N! \approx e^{N\ln N - N + \ln\sqrt{2\pi N}}$

Big O Notation

In computer science we usually only care about the asymptotic behavior of your function.
If runtime is 7N+3, the +3 doesn't really matter compared to 7N on large inputs.
Mainly it's the fastest growth rate term which most influences runtime of algorithm.
Big O notations give us a way to measure just this largest term.
Definition: 0(f(N)) is the class of functions g such that these exists a constant K such that $g(N) \leq Kf(N)$ for all $N \geq N_0$ for N_0 fixed.

Ex. 1: $7 \in 0(1)$
Proof: Choose No = 1, k = 7, then $N \geq 1$ $7 \leq 7.1$
Ex. 2: $7N + 3 \in O(N)$
Proof: Choose No = 1, k = 10, then $7 \bullet N + 3 \leq 10.N$ $N \geq 1$

END OF LECTURE AND SET #2 ***

LectureNotes

Fall 2000

Copyright 2000

PROGRAM IN COMPUTING 10B
PROFESSOR POLLETT
SET #3

OCTOBER 16, 2000

Remember HW2 is due Wed. Prof. Pollett will try to get students a practice midterm by Friday.

Last day-talked about math stuff
Today-will talk about big O-notation and recurrence relations.

Big O-notation
Definition: A function g(N) is in the class O(f(N)), i.e., $g(N) \in O(f(N))$, if there exists c_0, N_0 such that $g(N)<c_0 f(N)$ for all $N>N_0$

The point of big O-notation is it tries to capture the runtime of algorithms in a machine independent way. If we have an algorithm that performs $N^2 + N$ operations, the N^2 part of the algorithm will be most important to improve the speed of.
ex. show $N \in O(N^2-7)$
Let's take $c_0=1$ and take $N_0=3$
<u>Claim</u> for all $N>N_0$, $N<c_0(N^2-7) = N^2-7$
Proof (by induction):
Base case: N=4 then $4 \leq 16-7=9$ is true
Inductive step: Assume $N<N^2-7$ to prove that $N+1<(N+1)^2-7$
$(N+1)^2-7=N^2+2N+1-7$
    $> (N^2-7)+1$
    $> N+1$ by the inductive hypothesis
Thus $N<N^2-7$ for all $N>3$

General facts about O-notation
1. $O(1) \subseteq O(N)$
2. $O(N^k) \subseteq O(N^{k'})$ if $k' \geq k$
3. $O(f) + O(g) = O(f+g)$
4. $c\, O(f) = O(f)$ where c is a constant
5. $O(f)O(g) = O(f \cdot g)$
6. $O(f)+O(f)=2O(f)$

Ex. show
$x+x^2+x^3 \in O(x^3)$
$x+x^2+x^3 \in O(x)+O(x^2)+O(x^3)$ by 2
$O(x)+O(x^2)+O(x^3) \in O(x^3)+O(x^3)+O(x^3)$ by 2
$O(x^3)+O(x^3)+O(x^3)=3O(x^3)$ by 6
$3O(x^3) = O(x^3)$ by 4
Thus $x+x^2+x^3 \in O(x^3)$

Ex, $N^2 \notin O(N)$
Arguing by contradiction assume $N^2 \in O(N)$. Then there exists C_0, N_0 such that $N^2 \leq C_0 N$ for all $N>N_0$. Take $N=\max(C_0, N_0)+1$. Then $N^2=(\max(C_0, N_0)+1)^2 > C_0(\max(C_0, N_0)+1)=C_0N$ but that contradicts our assumption that $N^2 \leq C_0N$. Thus if we assume $N^2 \in O(N)$ that leads to a contradiction. Hence $N^2 \notin O(N)$.

Other useful facts:
1. $O(\log N) \underset{+}{\overset{\subset}{-}} O(N)$ ($E \underset{+}{\overset{\subset}{-}} F$ if E is a proper subset of F i.e. $E \subset F$, $E \neq F$)

2. $O(N^k) \underset{+}{\subset} O(m^N)$ for k, m fixed

Ex. $N^3 \in O(2^N)$ is true by 2 which says $O(N^k) \underset{+}{\subset} O(m^N)$ for K=3, m=2

How we can manipulate O in an expression (we can pretend it is like a constant).

Ex. show $\frac{N}{N+O(1)} = 1+O\frac{1}{N}$

$\frac{N}{N+O(1)}$ is a class of functions, $1+O(\frac{1}{N})$ is a class of functions.

To show $\frac{N}{N+O(1)} = 1+O(\frac{1}{N})$ we have to show (a) and (b) are true

(a) $\frac{N}{N+O(1)} \geq 1+O(\frac{1}{N})$ (b) $\frac{N}{N+O(1)} \leq 1+O(\frac{1}{N})$

(a) is easy to see

$$\frac{N}{N+O(1)} = 1+\frac{1}{NO(1)}$$
$$= 1 + \frac{1}{O(N)O(1)}$$
$$= 1+\frac{1}{O(N)}$$

Claim: $\frac{1}{O(N)} = O(\frac{1}{N})$

END OF LECTURE ***

OCTOBER 18, 2000

HW2 due today. HW3 on web. HW1 has been returned; 1 week period starting from 10/18/00 for regrades.

<u>Recurrence Relations</u>

We can often bound the number of operations for an algorithm on inputs of size N as a function of number of operations required on smaller instances.

<u>Ex</u> N number of copies made for n inserts into a dynamic array.

If resized array one bigger than old array . . .

$C_N = C_{N-1} + N \leftarrow$ recurrence relation

Recall $C_N = N + N - 1 + C_{N-2}$

$= N + N - 1 + N - 2 \ldots + 3 + 2 + 1$

$= \frac{N(N+1)}{2}$

$= \frac{1}{2}N^2 + \frac{1}{2}N$

$= 0(N^2)$

<u>Ex</u> If resized the array by doubling . . .

$C_{2N} = C_N + N$ with $C_1 = 1$, for N a power of 2.

So for C_m between N and 2N $C_m = C_{2N}$

So $C_m \leq C_{m/2} + m/2$

$C_m \leq C_{m/4} + m/2 + m/4$

$\leq C_{m/8} + m/2 + m/4 + m/8$

To solve for C_m in closed form . . .

Let $m = 2^n$, then

$C_m = C_{2^n} \leq C_{2^{n-1}} + 2^{n-1}$

$\leq C_2{}^{n-2} + 2^{n-1} + 2^{n-2}$

$C_2{}^n = (2^{n-1} + 2^{n-2} + \ldots + 1)(2-1)$

$= 2^n - 1$

$= 0(2^n)$

$= 0(m)$

Another very common recurrence relation is the Fibonacci sequence defined as

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ $n \geq 2$

To solve for F_n in closed form, let $\alpha \equiv \sum_{n=0}^{\infty} F_n x^n$. Then notice if you take $\alpha - \alpha x - \alpha x^2 = (F_0 + F_1 x + F_2 x^2 + \ldots) - (F_0 x - F_1 x^2 + \ldots) - (F_0 x^2 + \ldots.)$

$$= F_0 + (F_1 - F_0) + 0x^2 + 0x^3 + \ldots.$$
$$= x$$

So $\alpha(1 - x - x^2) = x \Leftrightarrow \alpha = \frac{x}{1 - x - x^2}$

Express $\frac{x}{1 - x - x^2}$ as a Taylor expansion and match F_i with ith coefficient you will get $F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$.

Solving for the Fibonacci sequence in closed form will not be on the test.
We'll only consider linear recurrences in this class.

Ex $C_N = 2_{N/2} + N \leftarrow$ only depends on one previous instance

This recurrence might come up with an algorithm like merge sort

Let $N = 2^n$

$$C_{2n} = 2C_2^{n-1} + 2^n$$
$$= 2(2C_2^{n-2} + 2^{n-1}) + 2^n$$
$$= 4C_2^{n-2} + 2^n + 2^n$$
$$= \underbrace{2^n + 2^n + \ldots + 2^n}_{n \text{ times}}$$
$$= n2^n$$

So O((logN)N) since $n2^n = (\log N)N$ because $N = 2^n$

Ex $C_N = C_{N/2} + 1$

Let $N = 2^n$

$$C_{2^n} = C_{2^{n-1}} + 1$$
$$= C_{2^{n-2}} + 1 + 1$$
$$= \underbrace{1 + 1 + 1 \ldots + 1}_{n \text{ times}}$$
$$= n$$

So $O(n) = O(\log N)$ because $N = 2^n$.

Let's start analyzing Algorithms

Ex Sequential Search

Input: a[l] . . . a[r], l < r

Problem: Find i such that a[i]==value. If it doesn't exist return -1.

```
int search (int a[ ], int value, int l, int r)
{  for(int i=l; i<=r; i++)
     if(value==a[i]) return i;
     return -1;
}
```

This algorithm's runtime will vary depending on the array.

$\boxed{1}\boxed{9}\boxed{2}$

If search for 1 it takes only 1 check.

Theorem: Let N be the size of array. If all locations equally likely, then sequential search examines on average $\frac{N+1}{2}$ numbers on successful searches and N numbers on unsuccessful searches.

Proof: Probability of looking at i numbers is $\frac{1}{N}$ but we have to do i steps in this case. Average number of steps to look is

$$1 \cdot \frac{1}{N} + 2\frac{1}{N} + 3 \cdot \frac{1}{N} + \ldots \frac{N}{N}$$

$$\frac{1}{N}\left(\sum_{i=1}^{N} i\right) = \frac{1}{N}\left(\frac{N(N+1)}{2}\right) = \frac{N+1}{2} = 0(N)$$

END OF LECTURE **

OCTOBER 20, 2000

HW3 is up. Practice midterm is up. HW2 solutions available later today.

Last day - Talked about recurrence relations. Analyzed average runtime for linear search.
Today - Analyze another algorithm argc, argv.
Then start talking about the list ADT.

More of analyzing algorithms

Consider the problem that we would like to make an array a such that a[i] = 1 it i is a prime and 0 otherwise.
(A prime is an integer, whole only divisions are 1 and itself)
We will use so called sieve of Erathothenes.
We'll produce an array for numbers less then...

```
#include<iostream.h>
Static const int N=1000 // N is how big a is.
int main( )
{    int i, a[N];
     a[0] = 0;
     a[1] = 0;
     for (int i=2; i<N; i++)
          a[i]=1;   //assume all numbers <N are prime to start
     for (int i=2; i<N; i++)
     {  if (a[i] == 1)
          {    for(int j=i; j*i<N, j++)
                    a[*j]=0;
          //if a[i] is still a prime, set all of its multiples to be not
          // prime
          }
     }
     for (i=2; i=N; i++)
          if(a[i] == 1) cout <<i<<"\n; //print out our primes
} //end main
```

pass	1	2	3	...
a[0]	0	0	0	...
a[1]	0	0	0	...
a[2]	1	1	1	...
a[3]	1	1	1	...
a[4]	1	0	0	...
a[5]	1	1	1	...
a[6]	1	0	0	...
a[7]	1	1	1	...
a[8]	1	0	0	...
a[9]	1	1	0	...

What's the runtime of this algorithm as a function of N?
1st line takes constant amount of time. O(1)Time
1st for loop we do ≈ N assignments
each assignment takes some constant amount of time NO(1) = O(N) time
2nd loop done N times and it's nested for loop does
first N/2 assignments then N/3 assignments then N/S assignments...
So 2nd for loop takes N/2+N/3+N/5+N/7+1/11+...

$$= \sum_{\substack{p \text{ prime} \\ p<1000}} \frac{N}{P}$$

$$\leq \sum_{i=1}^{N} \frac{N}{i} = N \sum_{i=1}^{1} \frac{1}{i}$$

$$\leq \int_{1}^{N+1} \frac{1}{i} di = (N+1)\ln(N+1)$$

Final for loop prints to screen N times $O(N) \in O(N\log N)$
$O(k) + O(N) + O(N\ln N) + O(N) = O(N\ln N)$

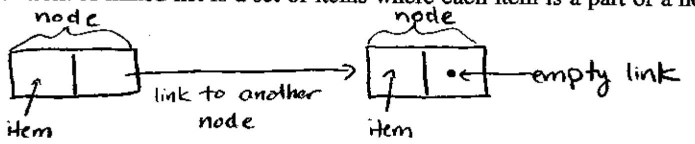
argc, argv

Say we want to run our program from command line and we want to pass a value to our program that sizes an array.
i.e., if under DOS or Unix

```
> myprog 1000 //supposed to tell your program that array size should be 1000.
//myprog
int main(int argc, char* argv[ ])
             ↑       ←
// # of command line inputs          //array of strings
{   int i, N = atoi(argv[i]); //atoi converts ascii string to integer
    int *a = new int[N]; //size array according to N.
}
    myprog 1000 hi
    // then argc = 3
    //argv[0]="myprog"
    // argv[1] = "1000"
    //argv[2] = "hi"
```

Linked list

Definition: A linked list is a set of items where each item is a part of a node that contains a link to a node.



END OF LECTURE AND SET #3 **

LectureNotes

Fall 2000

Copyright 2000

PROGRAM IN COMPUTING 10B
PROFESSOR [redacted] Pollett
SET #4

OCTOBER 23, 2000

midterm on Friday in class. Remember to bring photo id
Last day-started talking about linked lists. Talked about argc and argv
Today-Talk about linked lists

Typo correction

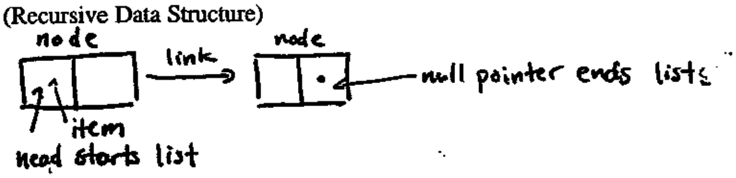
argc	argv
↑	↑
number of command line strings	array of command line strings

If command line was
myprog hi there

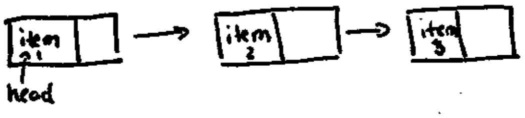
```
    //argc = 3
    //argv[0]=myprog
    //argv[1]=hi
    //argv[2]=there
```

Linked Lists
definition: a linked list is a set of items where each item is a part of node that contains a link to a node

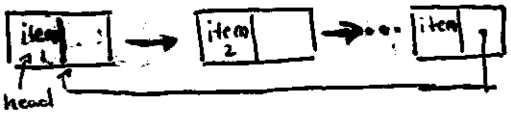
(Recursive Data Structure)



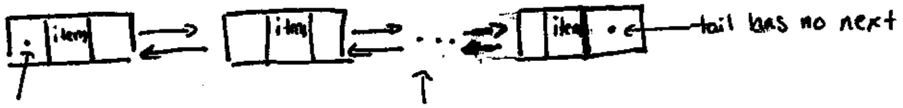
Some types of lists
1) standard singly-linked list



2) circularly linked list



3) doubly linked lists



head has no
previous node

each node has a pointer to a
next and previous node

Advantage: can back up in list
Disadvantage: more memory to implement

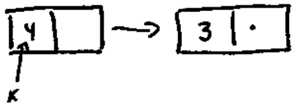
Implementing Lists
In class we will follow book and implement using structures. For HW use classes.

Code for a Node

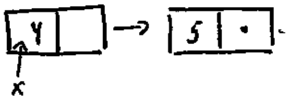
```
typedef int Item;   //now item type can be used (will mean same as int)
struct node {Item item, node *next};
                                       ↑
               //why recursive as refers to self.  Just an address so we don't need to know what a node is to define a node
typedef node *link;
```

ex. creating a list

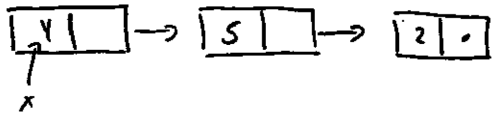
```
link head=new node(0, NULL);
link x;
x=head;  //now x points to same node as head
x→item=4;   //same as (*x) item=4 which sets item in x to be 4.
            //would also change head's item
x→item=new node(3, NULL);
```



```
x→next→item=5; //would change picture to...
```



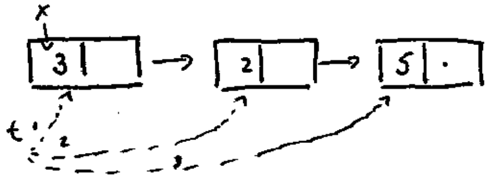
```
//can also do
x→next→next=new node(2, null);    //would look like
```



List operations

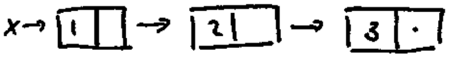
suppose we want to go through a list performing some operation on each item in list. Could do this with the following code...

```
for(link t=x; t != NULL; t=t→next)
     visit (t→item);     //visit is code we wish to perform on each item.
```

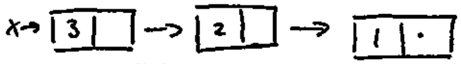


How to reverse the order of a list

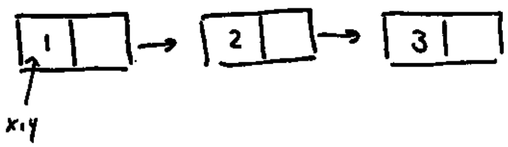
i.e. change



to

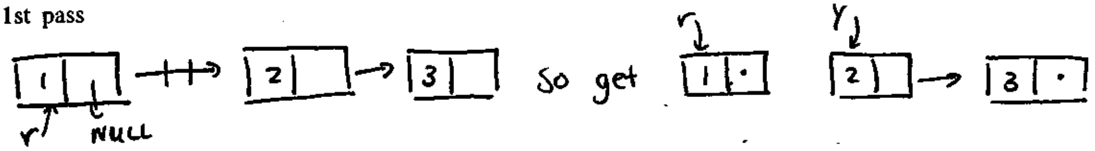


```
Link reverse (link x)
{
     link t, y=x, r=NULL;
          while (y != NULL)
          {     t=y->next;
                y->next=r;
                r=y;
                y=t;
          }
     return r;
}
```

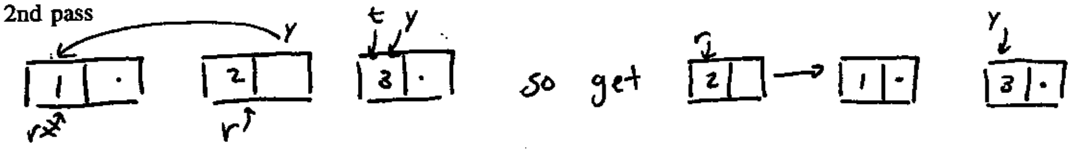


r=NULL

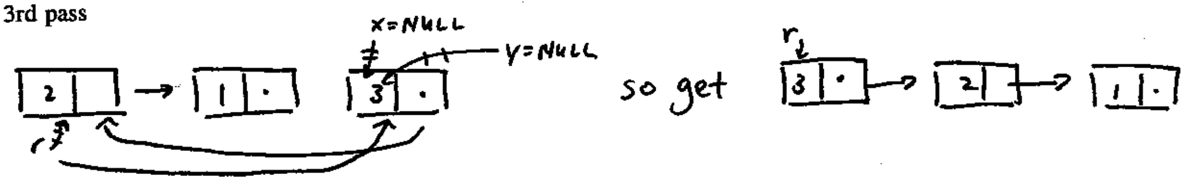
1st pass



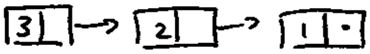
2nd pass



3rd pass



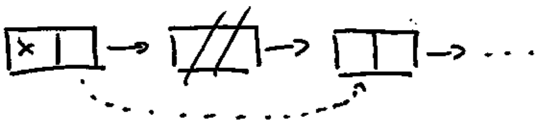
we now return r=



other operations

```
link head;
Initialized list: head=NULL;
Insert t after x: it (x==NULL)
     {     head=t; head->next=NULL;  }
     else
     {     t->next=x->next;
           x->next=t;
     }
test if empty: if (head==NULL)
```

Remove after x



```
t=x->next;
x->next=t->next;
delete t;
```

END OF LECTURE ***

OCTOBER 25, 2000

Midterm is on Friday. Closed books, closed notes. You must have photo I.D. Prof. Pollett will take 2 points off midterm test score if you begin test early or end test late. No baseball caps during testing.

Practice Midterm

(1) Define the following

(a) Copy constructor - is a member function of a class that takes as an argument an object, say o, of that class and produces a new object of this class, say O_2, such that O_2 has all its member data is the same as O. Copy constructor can explicitly invoked by the programmer or is invoked when returning objects of this class by value or when instantiating call-by-value object of a class.

```
MyClass fun( )
}    MyClass a;
     return a; // copy constructor evoked
}
Void fun2(MyClass 6)
```

```
{ }            copy constructor called
int main ( )
{    MyClass a; fun2(a); }
```

(b) friend function - a nonmember function of class which has access to a class' private data. Not called by runtime system.

(c) destructor - a member function of an object used to free up what memory it uses when we get rid of that object.
~ MyClass( ) is name of destructor for MyClass.
Can be called by runtime system. Ex: if object is a local variable and it goes out of scope of enclosing block.

```
    Ex.        { MyClass a;
               }
               // a's destructor would be called.
```

(d) private variable - member variable of a class only accessible by friend functions of class or by member of class.

(e) public variable - member variable of a class accessible from any function.

(2) Prove that NlogN$\in$ O(N^2)

Let $C_0 = 1$, $N_0 = 0$. Need to show NlogN $\leq C_0N^2$ for all $N > N_0$

Well NlogN $\leq N^2 \Leftrightarrow$ logN $\leq$ N
$\Leftrightarrow 2^{\log N} \leq 2^N$
$\Leftrightarrow N \leq 2^N$

We prove $N \leq 2^N$ by induction.

Basis step: N = 1 $1 \leq 2^1$ is true

Inductive step: Assume $N \leq 2^N$ is true to prove $(N+1) \leq 2^{N+1}$

$2^{N+1} = 2 \bullet 2^N = 2^N + 2^N \geq N + 2^N$ by the inductive hypothesis.
$\geq N + 1$ since $2^N \leq 1$ V $N \geq 0$

Therefore since $N \leq 2^N$ for all $N > 0$, that implies NlogN $\leq$ N. Thus if $c_0 = 1$, $N_0 = 0$
NlogN $\leq c_0N^2$ for all $N > N_0$. Then NlogN$\in$O(N^2) by definition.

(3) Show $N^2 \in$ O($N^{3/2}$)

Proof (by contradiction): Assume $N^2\in$O($N^{3/2}$), then there exist N_0, c_0 such that $N^2 \leq c_0N^{3/2}$ for all $N > N_0$. $N^2 \leq c_0N^{3/2} \Leftrightarrow N^{1/2} \leq c_0$. Take N=max ($N_0$, $(C_0 + 1)^2$) to get a contradiction.

(4) Say how to use #ifndet. #endif in header files.

```
#ifndef   My_h     //prevents the problem that header could be included
#define   My_h     //twice in your implementation or user files.
//your interface
#endif
```

(5) Explain what the this pointer is and give and example of use.

The this pointer is a pointer to the current object of which we're running the member function of.

```
Ex.      int main( )
         {   MyClass a;
             a.fun ( );
         }
         void MyClass::fun( )
         {     this →myfun( ); }
```

// In this example "this" would be the memory address of a.
// this pointer is useful for overloading = .

(6) Consider

$C_1 = 0$
$C_N = C_{N-1} + \log N$
Show $C_n\in$O(NlogN)
Well $C_n = (c_{n-2} + \log(N-1)) + \log N$
$= ((C_{N-3} + \log(N-2)) + \log(N-1) + \log N$

Expand N times to get C_n completely expanded as a sum of N things.
Each of which is less than logN.
so get $C_N \leq \underbrace{\log N + ... + \log N}_{N \text{ times}}$
$\leq$ NlogN for all N > 0
So $C_n\in$O(NlogN)

END OF LECTURE AND SET #4 **

LectureNotes

Fall 2000

PROGRAMMING IN COMPUTING 10B
PROFESSOR POLLETT
SET #5

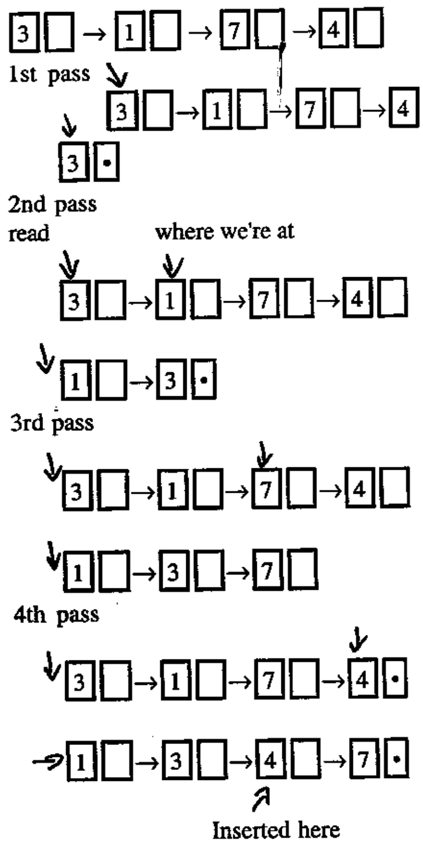
OCTOBER 30, 2000

HW3 due Friday 9:30 am

Today: Example of using linked-lists. Automatic type casting in C++. Templates

Application of Linked List

Sorting (Insertion Sort)



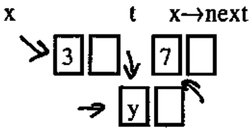
//Keep inserting item under consideration
//where it fits in sorted list.

For n node list we do insertion operation O(n) times. Each insertion takes O(n) time. So this is an O(n^2) algorithm.

Code fragment for insertion sort

```
node heada(0, NULL);
link a=&heada, t=a;;
for (int i=0; i<N; i++)                          //generates a list of length
     t=(t→next=new node(rand%1000, NULL));       //500 containing random numbers
     node headb(0, NULL);                         //less than 1000
     link u, x;
     link b=&headb;
for(t=a→next; t!=NULL; t=u);
{    u=t→next;
     for (x=b; x→next!=NULL; x=x→next)
```

```
    {   if (x→next→item>t→item)
        break;
    }
    t→next=x→next;
    x→next=t;
}
```



Automatic Type Casting

```
Consider float a=0.2;
        a=a+1;      //This is an int the way computer handles this is 1st typecasts
                    //to a float then adds
```

This is different from

```
    float operator + (float a, int b)
```

Can do this style also in C++ to allow type casting from say an int to your Class A just add a constructor which just takes an int.

```
class A
{   public:
        A(int i);       //used to convert from int to class A.
    .
    .
    .
```

So if we wanted to we could just define operator + for objects of type A

Templates

Often the case that we would like to write code that works on a variety of kinds of inputs.

Consider

```
void swap (int& var1, int& var2)
{   int temp;
    temp=var1;
    var1=var2;
    var2=temp;
}
```

If we changed int& var1 and int& var2 to type char& the same code would swap the two char's. Rather than have to re-enter same code for whatever type we want better to use templates

```
template<class TYPE>  //could be any C++ name
void swap (TYPE& var1, TYPE& var2)
    {   TYPE temp;
        temp=var1;
        var1=var2;
        var2=temp;
}
int main ( )
{   char a= 'A', b='B';
    swap (a, b);            //TYPE in this case is clear
    cout<<a<<b<<endl;       //BA
    int c=1, d=2;
    swap (c, d);
    cout<<c<<d<<endl;       //21
    return 0;
}
```

END OF LECTURE **

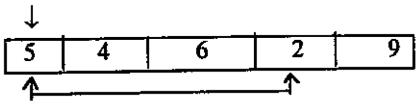
NOVEMBER 1, 2000

HW3 due Friday. Midterm average ≈ 15
Last day - started talking about templates.
Today - more templates

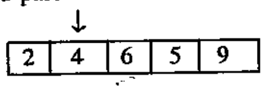
Selection Sort-with templates

Set up: have an array to be sorted. Have an index to keep track of what's already been sorted. We pick smallest item yet to be sorted and swap it with element in index and advance index.

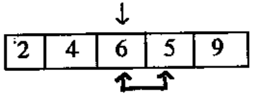
index (everything before index is sorted)



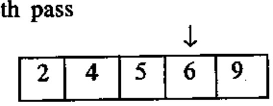
2nd pass



3rd pass



4th pass



done

We do O(N) swaps. Searching for smallest for a given swap look at O(N) elements. Sp total number of elements examined to perform all swaps O(N^2).

Above algorithm sorts "in-place" i.e. we do not need to create a temporary array to do sort.

Now let's write a generic sorting algorithm that works for any object for which < is defined.

Aside:

Definition: A generic function is one that can work on many types because defined using templates.

Will use swap values function from last lecture.

```
template <class T>
void sort(T a[ ], int size) // "size" is the size of array a.
{    int  indSmall;
     for (int i = 0; i < size-1; i + 1) // by time we get to size-1 array already sorted.
     {    indSmall = IndexSmallest(a i, size);
          swap_value (a[i], a[indSmall]);
     }
}
template <class T>
int IndexSmallest (const T a[ ], int start, int size)
{    T min = a[start];          //smallest so far
     int indMin = start;        //where smallest lives
     for (int i=start+1; i<size; i++)
     {    if (a[i] < min)
          {    min = a[i];
               indMin=i;
          }
     }
     return indMin;
}    //end of IndexSmallest

int main ( )
{    char c[5] = {'h', 'o', 'w', 'd', 'y'};
     sort (c, s);
     cout<<c[0]<<c[1]<<c[2]<<c[3]    //prints dhowy
          <<c[4]<<endl;
     return 0;
}
```

could also use sort for int's.

Templates with classes

```
Ex   template <class T>     //can have more than one generic type if so
                            //separate by comma
     class StoreObj         //stores one object of whatever type we want
     {    public:
            StoreObj (T ob1);
            T getObj ( );
        private:
            T object;
    };
//To write member functions
    template <class T>
    StoreObj::StoreObj (T ob1)
    {   object=ob1;}

    template <class T>
    T StoreObj::getObj( )
    { return object;}
//To use this class
Ex 1    StoreObj<int>    StoreInt(3);  //<int> says T is int
        cout<<StoreInt.getobj( )<<endl; //prints 3 to screen
Ex 2    StoreObj<char>   storeChar('a');
        cout<<storeChar.getobj( )<<endl;    //prints an a to screen
```

END OF LECTURE ***

NOVEMBER 3, 2000

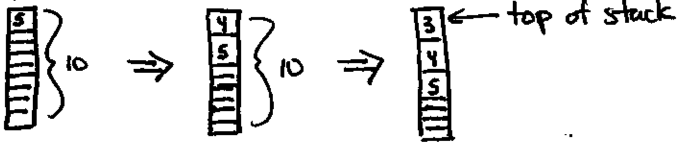
HW4 will be up later today

Last day - templates with functions and with classes
Today - Stacks (will use templates)

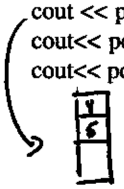
Stacks (since it's short, here's the interface...)

```
template <class Item>
class Stack
{   public:
        Stack(int s); //initializes the stack array to be size s.
        bool empty( ) const; //const means we won't modify private data.
        void push(Item item);
        Item pop( );
    private:
        Item *array;
        int maxSize, curSize;
};
```

```
//before look at code to implement a stack let's see how it's used/what's
//it good for.
Stack <int> s(10); //creates a stack of into of size 10.
push(5);
push(4);
push(3);
```

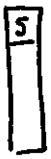


```
cout << pop( );
cout<< pop( );
cout<< pop( ) << endl;
```



3 returned

2nd pop return 4



4 returned and printed

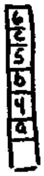
3rd pop returns and print

stack empty

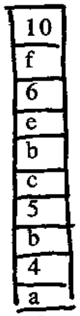
so above prints 345

Where are stacks used?
Any language after Algol uses runtime stack for function calls.

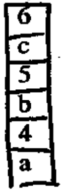
```
int main( )
{    int a = 4, b = 5, c = 6
     myfun( ); → we push
     :
}
void myfun( )
{    int e= 6, t = 10;
     myfun2( ); → after this call
}
```



on to stack that way when myfun( ) finished we can pop these values off the stack to find out what the values of the local variables of main were.



So after myfun2 ends pop e, f off stack to get their values and stack looks like



e would be set to 6 again, f would be set to 7.

One advantage to storing local variables in the stack during functions calls is that is allows functions to call themselves ,

```
int factorial(int n)
{    if (n = = 0) return l;
     return n*factorial(n-1);
}
```

(2) Used by compilers in syntax checking
Consider a language with the following pairs of tag

```
    'for

    endfor
```

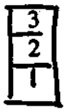
The following code would be legal

```
for
    for
    endfor
    for
    endfor
    endfor
```

```
    for       //but this would be illegal
    for
    endfor
    endfor
    endfor ← would try to pop an item from an empty stack.
```

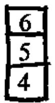
We could keep track of whether things nested corrected by whenever we see a for we push a symbol on the stack. And whenever we see an endfor insist we can pop an itme from stack. If can't pop then give an error.

(3) HW3
To read number push item onto stack
if had 1 2 3 would push



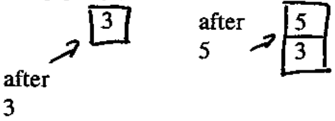
on stack

if second number was 456 on another stack would have

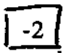


Now to odd, pop digit from top of each stack add them and any carry push result onto a third stack.
Now to point out sum keep popping till third stack empty each time printing a digit.

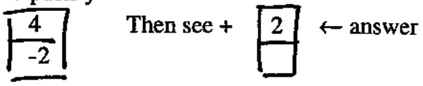
(4) reverse polish expression evaluation
5 4 + is same as 5+4
reverse polish notation
3 5 4 + - is 3-(5+y)
3 5 - 4+ is (3-5)+y
To evaluate such an expression with stacks
(1) Whenever see number push it on the stack
(2) Whenever see an n-ary operator pop n items from stack and evaluate accord to that operator push result onto stack.
After finished expression result is top of stack.
Ex. 3 5 - 4 +



see - pop 2 items from stack push -2 onto stack



now push y



END OF LECTURE AND SET #5 **

LectureNotes

Fall 2000

PROGRAM IN COMPUTING 10B
PROFESSOR POLLETT
SET #6

NOVEMBER 6, 2000

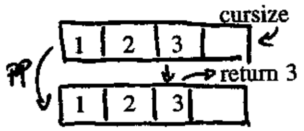
HW 4 is up. GetInput now returns as int.
HW3 solutions up, module commenting.

Last Day - was talking about stacks
Today - implementation of stacks. Make some remarks. Then talk about Queues and recursion.

Code for stacks

```
template <class Item>
Stack :: Stack(int s)
{    CurSize = 0;
     maxsize = s;
     array = new Item[s];
     if(array = = NULL)
     {    cout << "Out of memory\n";
          exit(1);
     }
} //end Stack

template<class Item>
Stack<Item>::~Stack
{ delete [ ] array;}
template<class Item>
bool Stack<Item>:: empty( )const
{ return curSize = = 0;}
template <class Item>
void Stack<Item>:: push(Item i)
{    array{cursize ++] = i;}  //on HW make sure array not full.
template <class Item>
Item Stack<Item> :: pop( )
{    return array[--cursize];}
```



Random Remarks:

(1) Code for implementation in book is in class definition

```
template<class Item>
Class Stack
{    //stuff
     Item pop( ) {  return array[ - -curSize];}

     // stuff
}
```

This is legal and sometimes useful if the implementation is really short. But if takes more than a couple lines to implement do as in class.

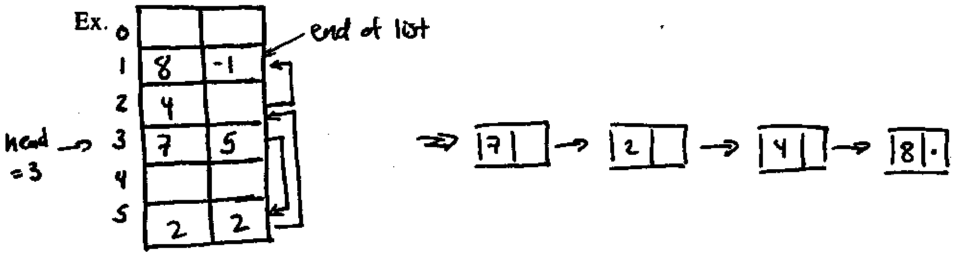
(2) Rather than implement using arrays could have implemented Stack using list. For example, to push an Item on stack create a new Node with that Item and insert at head of list.

(3) In above, pop, push, empty all take O(1) time to run.

(4) Sometimes hear term FILO used for stacks (First in Last out)

(5) Related to (2) we can in fact implement lists as 2-D arrays.

Ex.



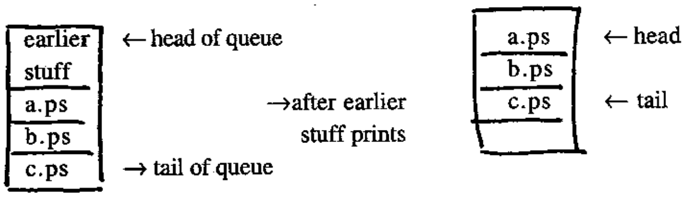
Queues - Like a stack except FIFO (first in first out)
Interface for Queues

```
class Queues
{     private:
      //implementation  Specific
      public:
            Queue (int s);
            ~ Queue( );
            bool empty( ); // is queue empty?
            void put(Item);
                  //put i at end of Queue.
                  Item get( );
                  //return first Item in Queue.
};
```

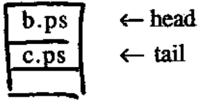
Ex. Printer Queues.
Suppose you want to print a bunch of files. In Unix, type

1pr a.ps <ret>
1pr b.ps <ret>
1pr c.ps<ret>

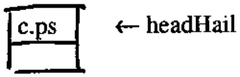
Can type this faster than it takes to print out a.ps. Also there may have been people before you printing stuff. What happens is your documents are placed on a print queue.


earlier stuff ← head of queue
a.ps
b.ps
c.ps → tail of queue
→after earlier stuff prints
a.ps ← head
b.ps
c.ps ← tail

After a.ps goes to printer

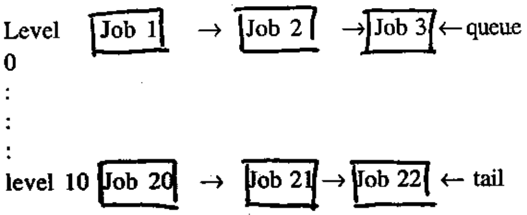

b.ps ← head
c.ps ← tail

Then b.ps goes to print.


c.ps ← headHail

Then Queue is empty. Basically printer call remove function of Queue.

Ex. 2: Operating systems sometimes use multilevel priority queues to schedule tasks. Each task has a priority. There is queue for each priority level giving which task to execute next.


Level 0 Job 1 → Job 2 → Job 3 ← queue
:
:
:
level 10 Job 20 → Job 21 → Job 22 ← tail

Jobs of level 0 execute first, then level 1, etc.
Among jobs of a given level acts like a queue, i.e., First in First out.

END OF LECTURE **
NOVEMBER 8, 2000

HW 4 due Tuesday

Couple of typos from last day.
When we have templates in driver file need to include .cpp file of implementation. So should probably put the .cpp file between #ifndef/#endif.
When we implement member functions we should do

```
template <class Item>
Stack<Item>::Stack (int s)
{    //code
}
```

Today we will talk about recursion
Definition: A recursive algorithm is an algorithm which calls itself.

<u>Ex</u>

```
int factorial (int N)
{    if (N==0) return 1; ← Base cases or stopping case
     return N*factorial(N-1);      //**
}
```

<u>Ex</u>

```
factorial (3)
     returns 3*factorial (2)
               returns 2*factorial (1)
                         returns 1*factorial (0)
                                   returns 1
```

So get 3*2*1*1 = 6 returned

Saw before that function calls are implemented with stacks. So functions that call themselves okay. What is the advantage of having recursive functions?

1. Very useful in coming up with correctness proofs of your algorithms.
2. Generally, easier to get a recurrence relation that bounds runtime of your algorithm.
3. Much easier to write compilers using recursive descent methods than iterative ones.

What are the disadvantages?
Since use runtime stack takes more space to run a recursive program.
Also calls to runtime stack generally slower than incrementing a counter in an iterative program.

<u>Ex</u> proof of correctness of factorial
<u>Claim</u> for all n factorial (N) returns N!
Proof by induction
Base Case: When N==0 factorial (N) returns 1 and 0! = 1, so works.
Inductive Step: Assume factorial (N) = N! to prove factorial (N+1)=(N+1)!
Consider factorial (N+1), line ** will be executed.
So (N+1)*factorial(N) returned. Factorial(N) = N! By the inductive hypothesis.
So factorial (N+1)=(N+1)*factorial(N) = (N+1)*N! = (N+1)!

Number of sub calls to factorial on factorial(N), Subcalls(N) = 1+Subcalls(N-1) Subcalls(0)=0. Subcalls(N) is a recurrence relation and Subcalls(N)=N.

<u>Ex</u> Let's try to write a recursive function which computes x^n for $n \geq 0$

```
double pow(double x, int n)
{    if (n==0) return 1;
     return x*pow(x, n-1);
}
```

As in factorial case pow(x, n) makes O(n) subcalls to compute n.

$$x^n = \begin{cases} (x^{n/2})^2 \text{ if n is even} \\ (x^{n/2})^2 * x \text{ if n is odd} \end{cases}$$

Using this can get an algorithm which takes O(log N) many subcalls.

```
Double pow2(double x, int n)
{    double temp;
     if (n==0) return 1;
```

```
    temp=pow2(x, n/2);    //n/2 rounds down
    if(x%2==0) return temp*temp;
    return temp*temp*x;
}
```

Why did I use temp rather than pow(x, n/2)*pow(x, n/2)*x?
To avoid recomputing pow2 twice which would produce no savings over previous algorithm.

Some remarks on recursion:
If don't have a good base case can get infinite loops.

```
void bob()
{    bob(); }  //will go till runtime stack overflows.
```

END OF LECTURE AND SET #6 ***

LectureNotes Fall 2000

PROGRAM IN COMPUTING 10B
PROFESSOR POLLETT
SET #7

NOVEMBER 13, 2000

Hw3 will be returned by 3. Hw 5 up tomorrow.

Last day-recursive functions
Today-more recursive functions, recursive member functions, function pointers, memoization.

Recursive member functions (a member function which calls itself)

```
ex. class MyNum
    {   public:
            //some methods
            double pow(int n);
        private:
            double x;
    }
    double MyNum::pow(int n)
    {   int temp;
        if (n==0) return 1;     //example of recursion in a member
        temp=pow(n/2);          //function
        if (n%2==0) return temp*temp;
        else return temp*temp*x;
    }
```

Some recursive list operation

```
int length(link x) ← computes length of list
{   if (x==NULL)
        return 0;
    return 1+length(x→next);
}
```

```
void traverse(link h, void visit(link))     hey, we're passing a function
//this function goes through each node of the list and does a visit operation on each node.
{   if (h==NULL)
        return;
    visit (h);     //do whatever visit does on h
    traverse(h→next, visit);
}
```

since recursive call last statement in block of code, this function is called <u>tail recursive</u>. A tail recursive function can easily be replaced by a for loop and some compilers will do this automatically.

Back to fact we passed a function as a argument...
Just like any variable each function you write has an address in memory so you can have a pointer to this address and use this pointer to run the code at that address.

```
ex. void (*myfnptr)(link h);
*myfnptr is a pointer to function which takes a link as its argument and returns a void.
void myvisit(link h)
{ cout << h→item << endl; }
myfnptr=&myvisit;     //in main or some function we would write this
//now myfnptr points to where compiled my visit code is.
(*myfnptr)(mylist);     //where mylist is a link calls the function myvisit on mylist
```

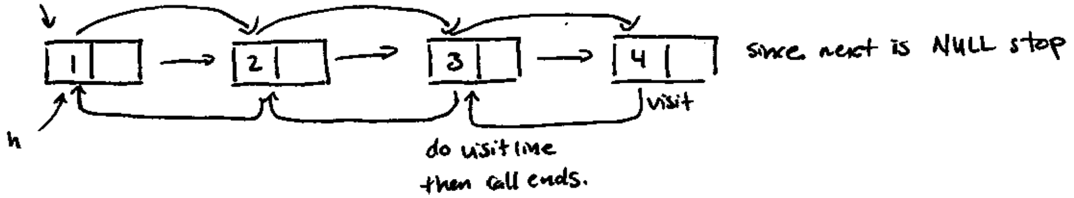
To call traverse could do:
traverse(mylist, myvisit); //can't remember may need &

Can "fake" classes in C using functions pointers as member variable of structures.

Let me now go back to writing one more recursive version of a list operation

```
//traverse a list from the end to the start
void traverse R(link h, void visit (link))
{    if (h==NULL) return;
     traverse R(h→next, visit);
     visit(h);
}
```

//this code uses the runtime stack to store list in reverse order first then visits each node in this reversed list.



Divide and conquer
Idea: recursively split problem into two subproblems of roughly same size, do each subproblem, then combine the result.
Divide and conquer to find max item it an array.

```
template <class Item>
Item max(Item a[ ], int l, int r)
//precondition: < defined for Items
{    if (l==r) return a[l];
     int m=(l+r)/2;      //midpoint of area in which searching for a maximum
     Item u=max(a, l, m);
     Item v=max(a, m+1, r);
     if (u>v) return u;
     else return v;
}
```

END OF LECTURE **

NOVEMBER 15, 2000

HW5 is up. Will have HW4 solutions up later today.

Last day - talked about Divide and conquer algorithms
Today - memoization, graphs/trees

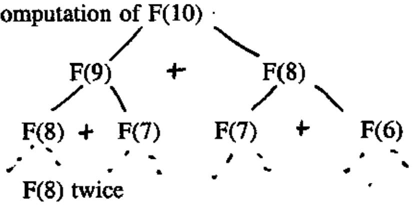
Dynamic Programming/Memoization = approach to solving recursive problems where we store subproblems have already taken care of.

Consider the following very inefficient algorithm to compute Fibonacci numbers:

$F(0) = 0$
$F(1) = 1$
$F(n) = F(n-1) + F(n-2)$

```
     int F(int i)
{    if(i<1) return 0;
     if(i = = 1) return 1;
     return F(i-1) + f(i-z);
}
```

This algorithm will take exponential amount of time to compile F(i).
To see this consider computation of F(10)



Notice we compute F(8) twice

F(7) three times
F(6) 5 times

We recompute F(1), F(0), . . ., F(10) many times since fibonacci sequence is exponential in i. We have to do exponentially many recomputations of leaves. However, not too hard to compute Fibonacci sequence in linear time iteratively.

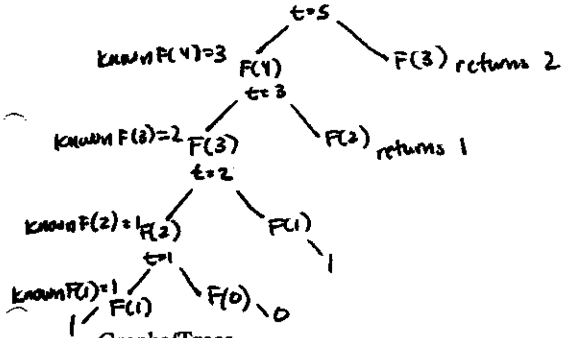
```
int maxN = 1000,  F[maxN];
F[0] = 0; F[1];
for (i = z; i < maxN; i++)
F[i] = F[i-1] + F[i-2];
//starts at F[0], F[1] and computes up to F[maxN-1].
//This is called bottom-up approach. Recursive approach work, from
//F(i) to F(1), F(0). This is called top-down approach.
```

Can we make the top-down approach as efficient as bottom-up approach? Yes.
Solution is to memoize i.e., store subcomputations.

```
int  F(int i)
{     static int knownF[maxN];   //will store values we've already computed.
                                 //Static local variables return their values
                                 //between function calls.
      if(knownF[i] ! = 0) return known F[i];
      int t;
      if(i < ) return 0;
      if(i = = 1) return known F[i] = 1;
      if(i > 1) t = F[i-1] + F[i-2];
      return known F[i] = t; //store t then returns its value
}
```

known F(5) = 5, 5 returned

Consider F(5)

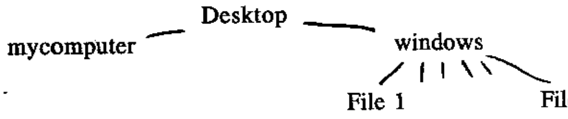


so this algorithm is linear in n.

Graphs/Trees

Trees - useful in describing dynamic properties of algorithms. For instance, Fibonacci program above. Also useful as date structures to organize information.

Ex. (1) File directories: can think of desktop as root of tree.
Folders are internal nodes. Files are leaves.



(2)Search trees: a way to organize date say in a dictionary, database, for quick retrieval.
(3) Sorting algorithms

Uses of graphs

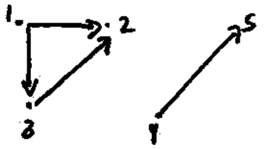
Useful to represent any kind of network (internet, neural net, cellular automata). In graphics all objects are approximated with polygons which are graphs.

Formal Definitions

Definition: A (directed) graph is an ordered pair G = <V,E>

V-a set of vertices

$E \subseteq V \times V$ - a set of edges
↑
ordered pairs in V.



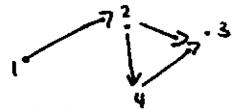
G = <{1,2,3,4,5}, {(1,2),(1,3),(3,2),(4,5)}>
↑
G represent picture.

If we had written edge as {3,2} rather than (3,2) and similar for each edges, we'd get undirected graph.

END OF LECTURE **

NOVEMBER 17, 2000

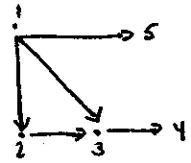
HW5 due Wednesday not Monday after Thanksgiving
Last day - we defined graph
Today - talk about trees. Draw some pictures, define things.
Recall a graph was a pair G = <V,E>

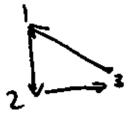


Need a couple definitions to talk about trees

Definition: a sequence of vertices connected by edges leading from one vertex to another vertex with no vertex appearing twice is called a simple path. A simple path with same start and end vertices is called a cycle.

Ex





(1, 2, 3, 4) is a simple path
(1, 4, 3) not a simple path since no edge between 1 and 4

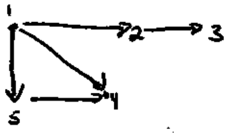
(1, 2, 3, 1) is a cycle.

Definiton: A graph is connected if any two vertices are connected by a simple path.

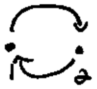
Ex. To avoid having to make doodles

to say a vertex is connected to itself we consider singleton sequences as legitimate simple paths

(1) is connected

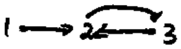


is not connected



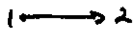
is connected

Ex.



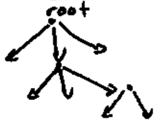
can't go 2 to 1, not connected

Ex.

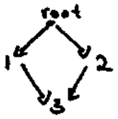


no can't go 2 to 1

Definition: A rooted tree is a graph with a vertex called the root such that there is exactly one simple path from the root to any vertex and any edge the graph belongs to one such path.

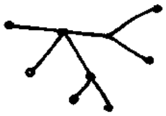


← is a tree

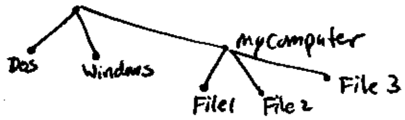


not a tree since there are 2 paths to 3.

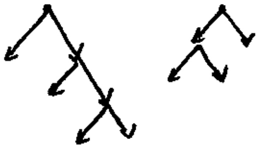
If take a rooted tree and "forget" directions then you have a tree (sometimes called a free tree)



A tree is always connected
Ex.

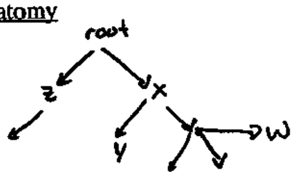


Suppose we have



not a tree but this graph contains two trees.
so a graph made up of trees called forest

Tree Anatomy



x,z are siblings
both children of root
x is parent of y
y is child of x
x is an ancestor of w.

Let's talk about implementing trees
template <class T>
struct node { Item i; } node *l, *m, *r;}
// these addresses
// would be stored in consecutive locations in memory

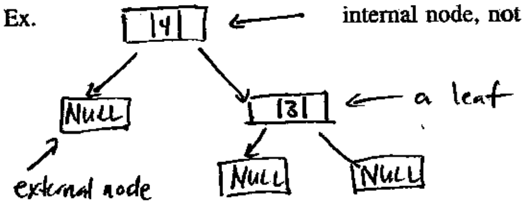
so have fixed some order on subtrees of a give node.

(Aside: A subgraph of graph G = <V,E> is G'= <V',E'> such that V' $\subseteq$ V, E' $\subseteq$ E. A subtree is a subgraph of a tree which is a tree).
An ordered tree is a rooted tree in which children of every vertex are given in some order. (So corresponds to how we implement trees with structures)
Also how we draw trees

left ← →right
So can order node by how for right they are.
Can tweak above structure if want two children or more than 3.
template <class Item>
Struct node {Item i; node *l, *r;}

If we wanted more than three
template <class Item>
struct node { Item i; node *children [ ]; } however many we want
Definition: M-ary tree is an ordered tree such that every vertex has 0 or M many children. In case M = 2, called binary tree. Now 0 children case is implemented by a pointer to a NULL node. These are called external nodes. Nodes which aren't external are internal. At leaf for an ordered tree is an internal node whose children are all external (A leaf in a rooted tree is a node without children)

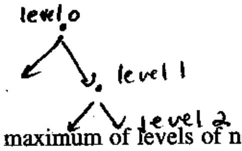
Ex.



internal node, not a leaf

This is a binary tree

Definition: The level of a node in a rooted tree is the number of edges in a simple path from root to node.



Height of tree is maximum of levels of nodes in the tree
Ex. This tree has height 2.

END OF LECTURE AND SET #7 ***

LectureNotes

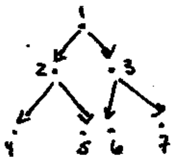
Fall 2000

PROGRAMMING IN COMPUTING 10B
PROFESSOR POLLETT
SET #8

NOVEMBER 20, 2000

Last day-was talking about trees
Today-more trees, tree traversals.

Given a rooted tree

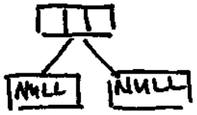


the level of a node is the length of simple path from the root to the node.

level of node 3 is 1
root is level 0
level of node 5 is 2
Height of tree is the maximum level in the tree. For above tree is 2.
Height useful to bound search times for items in trees.

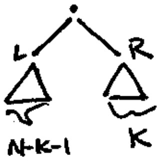
Definition: the Path Length of a tree is the sum of the levels of all the tree's nodes.
ex. for above tree 0+1+1+2+2+2+2=10

Internal path length-sum levels of internal nodes
External path length-sum levels of external nodes



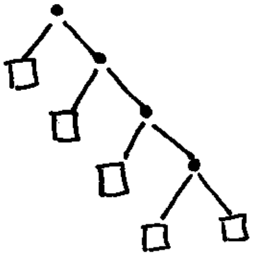
IPL=0
EPL=1+1=2

Theorem A binary tree tree with N internal nodes has N+1 external nodes.
Proof (by induction)
If tree is just a root which is a null pointer. Have 0 internal nodes; 1 external node so works.
Assume true for binary trees with <N nodes. Consider a tree which N internal nodes. Left subtree has N-K-1 internal nodes. Right substree has K<N internal nodes.

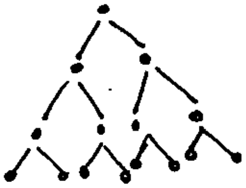


By the induction hypothesis the left subtree has N-K-1+1=N-K external nodes right subtree has K+1 external nodes. Adding we get N-K+K+1=N+1. QED.

Theorem The height of a binary tree with N internal nodes is at least logN and at most N-1.
Intuition

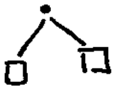


Height=3 has 4 internal

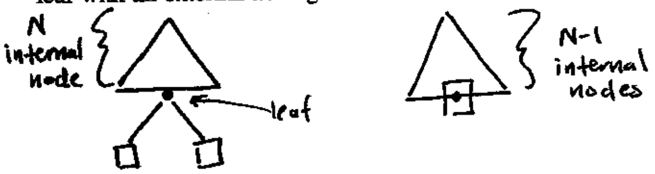


number of internal nodes ~1/2 nodes in tree log of # nodes in tree gives height.

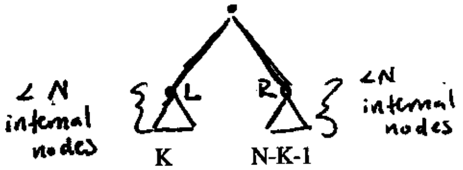
Proof (by induction) for the tree with 1 internal node is true
Height is 0=log1=1-1



Suppose true of trees with <N nodes. Take any tree with N internal nodes. It must have at least one leaf (prove this by induction). Replace this leaf with an external node gives a tree with N-1 internal nodes.



By induction hypothesis, this latter tree has height <N-2. So former is at most one taller so has height <N-1.
For⌈logN⌉case, consider root of tree.



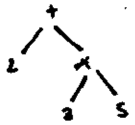
K internal nodes N-K-1 internal nodes

So can apply induction hypothesis to two subtrees.
Height of the whole tree is max{logK+log(N-K-1)}+1

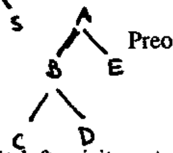
If try to make this as small as possible K = N/2. So height is at least $\log\left(\frac{N}{2}\right) + 1 = \log N + \log\frac{1}{2} + 1 = \log N$.

Tree traversals

ex. 2+(3*5)



could be useful to go through the nodes of tree in some order to calculate the value of this expression.



Preorder traversal visit current node then visit left then visit right ABCDE

Inorder-visit left, visit root of current tree, visit night subtree
CBDAE
Postorder traversal-visit left, visit right, visit root.
CDBEA

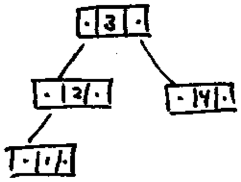
END OF LECTURE **
NOVEMBER 22, 2000

Last Day - we proved some facts about trees we talked about preorder, inorder, postorder tree traversals.
Today - look at code for above and other algorithms about trees.

Code for preorder traversal (using recursion) //visit Root - Left - Right
void preorder (link h, void visit(link))

```
{    if(h = = NULL) return;
     visit (h);
     preorder (h→l, visit);
     preorder(h→r, visit);
}
```

How to "hardwire" a tree to test out this function.
Let's say want to create the tree,



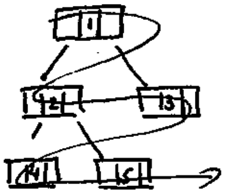
Using our struct node for binary trees where link is a typedef for node*

```
node root, left, right, lleft;
link h = &root;
root.item = 3;
root.l = &left;
root.r = &right
left.item = 2;
left.l = &lleft;
left.r = NULL;
right.item = 4;
right.l = right.r = NULL;
lleft.item = 1;
lleft.r = lleft.l = NULL;
preorder(h,vis);
// where vis is
void vis(link h)
{cout << h → item << endl;}
```

Output

```
3
2
1
4
```

Level-order traversal   //useful for breath first search



traverse each level in turn left to right 1 2 3 4 5

Unlike other three traversals which were coded using recursion (and hence implicitly used stack) we will code this iteratively using a queue.
Let's assume we've already coded a template class for Queue.
To do level order traversal...

```
void levelorder(link h, void visit(link))
{    Queue<link> q(max);    //max is some static constant
     //queue that stores links
     q.put(h);
     while(!q.empty( ))
     {    visit(h = q.get( ));
     // get head of queue. Set h equal to this visit h.
          if (h→l ! = NULL) q.put(h→l);
          if( h→r ! = NULL) q.put(h→r);
     }
}
```

Suppose h was

Suppose visit is vis given above
First put 3 on queue
Second remove 3 print it
        put 2 4 on queue
Third remove 2 print it
        put 1 on queue
Fourth remove 4 print it
        nothing to put on queue
Fifth remove 1 print it
        done
Print   3
        2
        4
        1

Simple algorithm to print out a tree.

```
void printNode(Item x, int h)
{    for(int i=0; i < h; i++)
          cout << "   ";
     cout << x << endl;
}

void show (link t, int h)
{    if (t = = NULL)
     {    printNode ('*', h);
          return;
     }
     show(t → r, h+1);
     printNode (t → item, h);
     show (t→l, h+1)
}
```

The function show prints out tree t h spaces over

```
                *
            4       *
        3           *
            2       *
                     1    *
                          *
```

END OF LECTURE AND SET #8 ***

LectureNotes

Fall 2000

PROGRAMMING IN COMPUTING 10B
PROFESSOR POLLETT
SET #9

NOVEMBER 27, 2000

HW4 returned by tomorrow. This Friday practice final on the web.

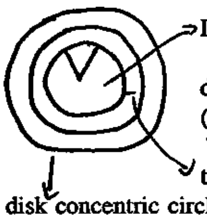
Last Day - talked about tre^ traversals and algorithms based on trees.
Today - sorting algorithms.

Sorting

Have a collection of objects for which operator < defined (also have operator =, == defined). Would like to put objects in order.

Ex FUDGE
↓
DEFGU

Today will consider algorithms which are $O(N^2)$ time to sort N items.
For large data sets best algorithms are O(NlogN).
However, the algorithms we consider today will be easy to write and for small data sets work well.
Some considerations about sorting algorithms.
Internal vs. External Sorting
In internal sorting data to be sorted fits entirely in memory.
External sorting involves leaving some of data on disk or tape during sorting process.



⇒Disk access will be slow part of your algorithm

data from disk read in blocks of some fixed size (512 bytes)
Want to minimize these block accesses. (HW 6)
tracks broken into sectors ≈ blocks
disk concentric circles called tracks

Let's take a look at a first sorting algorithm.

Insertion Sort

Idea: taking first unsorted item and inserting in correct location in sorted part
↓

Ex FUDGE
↓
FUDGE
compare these two. Since already sorted, don't do anything.
↓
FUDGE ⇒ FDUGE ⇒ DFUGE
↓
DFUGE ⇒ DFGUE ⇒ DFGUE
↓
DFGUE ⇒ DFGEU ⇒ DFEGU ⇒ DEFGU ⇒ DEFGU done

On a list of N items we advance i, O(N) times.

On each advance of i need to do an average O(N) comparisons /swaps.

So algorithm is $O(N^2)$

Code for Insertion sort

```
template <class Item>
    void exch (Item &A, Item &B)
    {    Item t=A; A=B; B=t;}   //swaps A and B. Note =
                                //overloaded.
template <class Item>
    void compexch (Item &A, Item &B);
    {    if (B<A) exch(A, B); }  //note need < defined.
template <class Item>
    void sort (Item a[], int l, int r)    //sort between l and r.
    {    for (int i=l+1; i<=r; i++)
              for (int j=i; j>l; j--)
                   compexch (a[j-1], a[j]);
    }
```

Above is a so-called nonadaptive sorting algorithm i.e., operations we did, did not depend on how the original array was ordered.
(If array was already ordered would do same comparisons. Swapping of course depends on array).

Adaptive - comparisons made during sorting depend on how the data set ordered.

Some more issues to consider when sorting . . .
Consider
struct person {char fname[20], lname[20];};
Tom Philben
Bob Parker ←suppose operator < just compares last name
Regis Philben

Then both

1.		2.
Bob Parker		Bob Parker
Tom Philben	&	Regis Philben
Regis Philben		Tom Philben

should be considered sorted.
1. Preserves the fact that Tom Philben occurred before Regis Philben in original list.
2. Does not preserve this property.
Sorting algorithm which preserves this property is called stable
Insertion sort is stable.
Remember selection sort.

END OF LECTURE **************************

NOVEMBER 29, 2000

Hw6 up. Bonus up by later today. Practice final up Friday. For honors section we will meet on Monday at 2 and discuss bonus.

Last day-talked about insertion sort
Today-making insertion sort faster and talk about bubble sort.

Remember selection sort for final.
Ex Selection Sort

| 5 | 3 | 9 | 7 |

Find minimum in unsorted part
i.e. 3

| 3 | 5 | 9 | 7 |

swaps minimum value with left most to be sorted value

Sorted

| 3 | 5 | 9 | 7 |

find min
swap with 5
sorted

| 3 | 5 | 9 | 7 |

find min
7 swap with 9
sorted

| 3 | 5 | 7 | 9 |

all sorted

| 3 | 5 | 7 | 9 | done

Let's go back to insertion sort and try to make it faster. Suppose already sorted to here in array

Sorted
Consider | 2 | 3 | 5 | 6 | 4 |

i.e. | 2 | 3 | 5 | 4 | 6 |

| 2 | 3 | 4 | 5 | 6 |
compares does nothing

| 2 | 3 | 4 | 5 | 6 |
compares does nothing

1) Only need to keep comparing values and swapping till find a value less than 4. Actually, we don't need to keep doing swaps. We store 4 then move everything after the 3 right by one.

| 2 | 3 | 5 | 6 | 4 | store 4

| 2 | 3 | 5 | 6 | 6 | move 6 right by 1 since greater than 4

| 2 | 3 | 5 | 5 | 6 | move 5 right by one

4>3, so then place 4 in this opened slot

| 2 | 3 | 4 | 5 | 6 |

this saves us the time need in swap to store things in a temporary variable.
For this to work need to know that there was some value left of 4 initially that is less than 4. Otherwise, could end up going off left of array.
To handle this we first find the least item in the array and put it in the zeroth position. This smallest value is sometimes called a sentinel
i.e. it's a value that guards so we don't leave our data structure. In this case, the array.

Revamped code for insertion sort

```
template<class Item>
void insertion(Item a[ ], int l, int r)
{    int i;
     for (i=r; i>l; i--)
          compexch(a[i-1], a[i]); //set up sentinel
                                  //smallest value in a[l]
     for (i=l+2; i<=r; i++)
     {
          int j=i; Item v=a[i];     //v store a[i] till we find where to
                                    //insert it
          while (v<a[j-1])
          {    a[j]=a[j-1];
               j--;
          }
          a[j]=v;
     }
}
```

Save ourselves a couple of assignments each loop over old program since don't use temporary variable. Turns out this program is roughly twice as fast. Unlike our previous version of insertion sort this version is adaptive, i.e., comparisons made does depend on original order of array.

Bubble sort
Start with
Start at this side

| 6 | 4 | 9 | 5 |

compare current value at index with value at index-1, swap if smaller. Keep cycling through array till get to left hand side.

| 6 | 4 | 5 | 9 |

↑

no change

| 6 | 4 | 5 | 9 |

swap

| 4 | 6 | 5 | 9 |

↑

4 is sorted

Sorted

| 4 | 6 | 5 | 9 |

don't do anything

| 4 | 6 | 5 | 9 |

swap

sorted

| 4 | 5 | 6 | 9 |

do nothing

sorted |

$\boxed{4|5|6|9}$

sorted |

$\boxed{4|5|6|9}$

END OF LECTURE

****

DECEMBER 1, 2000

Practice Final is up. Final is in Moore 100 on December 10 (Sunday), 3-6

Last Day – Talked about speeding up insertion sort, bubble sort
Today – bubble sort, merge sort

<u>Bubble Sort</u>

Idea: go in passes, right to left "bubbling" smallest unsorted value to the left.

<u>1st pass</u>

nothing is sorted — start at right

$\boxed{6|4|9|5}$

5 smaller so swap

$\boxed{6|4|5|9}$

4 smaller so don't swap

$\boxed{6|4|5|9}$

4 smaller so swap

sorted |

$\boxed{4|6|5|9}$

Now 4 is in correct position

<u>2nd pass</u>

sorted |

$\boxed{4|6|5|9}$

5 smaller so don't swap

sorted |

$\boxed{4|6|5|9}$

5 smaller so swap

sorted |

$\boxed{4|5|6|9}$

Now 5 is in correct position

<u>3rd pass</u>

sorted |

$\boxed{4|5|6|9}$

don't swap since 6<9

sorted |

$\boxed{4|5|6|9}$

4th pass pass done

How many comparisons made by Bubble sort to sort N items?
We make N passes through array
1st pass make N-1 comparisons
2nd pass make N-2 comparisons
3rd pass make N-3 comparisons
⋮ ⋮ ⋮
last pass 0 comparisons
Then total number of comparisons $\Sigma i = 0 + 1 + \ldots + N-1 = N(N-2)/2$

So we make $O(N^2)$ comparisons
So Bubble sort is $O(N^2)$ algorithm

Code for Bubble Sort

```
template <class Item>
void bubble (Item a[ ], int l, int r)
{ for(int i=l; i<=r; i++) // how many passes we've made i.e., where
                          // the sorted marker is.
    {  for (int j=r; j>i; j--)  // does bubbling to the left for a given
                                //pass
          compexch (a[j-1]; a[j]);
    }
}
```

Let's look at a faster sort algorithm

<u>Merge Sort</u>

A divide and conquer algorithm
So if want to sort $\boxed{5|2|9|6|7|3}$
split into two halves
$\boxed{5|2|9}$ $\boxed{6|7|3}$
sort these two halves and then "merge" the results.
So would first sort left half
$\boxed{5|2|9}$
split into two halves – sort left half
$\boxed{5|2}$ $\boxed{9}$
split into halves
$\boxed{5}$ $\boxed{2}$
$\boxed{5}$ ← sorted
sort right half of $\boxed{5|2}$
$\boxed{2}$ ← sorted
then merge the results
↓ ↓ output array
$\boxed{5}\boxed{2} \rightarrow \boxed{2}$
↓ ↓
$\boxed{5}\boxed{2}$ outside of region so copies → $\boxed{2|5}$
Now sort right hand of $\boxed{5|2|9}$
$\boxed{9}$ ← sorted
Now merge
↓ ↓
$\boxed{2|5}$ $\boxed{9} \rightarrow \boxed{2}$
↓ ↓
$\boxed{2|5}$ $\boxed{9} \rightarrow \boxed{2|5}$
↓ ↓
$\boxed{2|5}$ off the array $\boxed{9} \rightarrow \boxed{2|5|9}$
so copy what's left of other

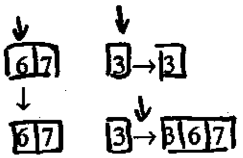
Now done sorting left half of $\boxed{5|2|9|6|7|3}$
so sort right half
$\boxed{6|7|3}$ split in two then sort left and right

$\boxed{6|7}$ split again

$\boxed{6}$ ← sorted

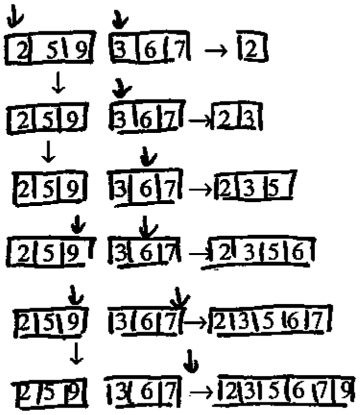
so sort right half of $\boxed{6|7}$
$\boxed{7}$ ← sorted
↓ ↓
$\boxed{6}$ $\boxed{7} \rightarrow \boxed{6}$

↓ ↓
$\boxed{6}$ $\boxed{7} \rightarrow \boxed{6|7}$
Now sort right half of $\boxed{6|7|3}$
$\boxed{3}$ → sorted
Merge results



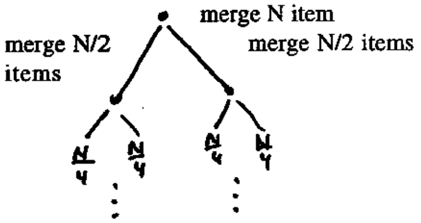
since outside of sorting region, can just copy

Now Merge



← sorted array

Notice when merge two halves which have combined a total of N elements, it is an O(N) operation.
So now let's look at recursion tree for sorting N items



log N many levels of merges on each level do 0(N) many operations. So total runtime O(N)•logN = O(NlogN)

END OF LECTURE AND SET #9 *****************

LectureNotes *Fall 2000*

PROGRAMMING IN COMPUTING 10B
PROFESSOR POLLETT
SET #10

DECEMBER 4, 2000

HW5 solutions up – will go over practice Final on Friday

Last Friday – talked about MERGESORT
Today – will give code of mergesort start talking about Hash tables

Remember how merge Sort worked:
To sort

9	3	8	2	7	1

Split into two halves

3	8	9

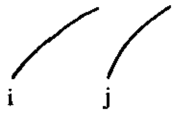
1	2	7

sort both halves
Merge two halves to get sorted array

1	2	3	7	8	9

Let's write code for MergeSort
First we'll write code for doing the merging

```
template < class Item>
void merge (Item a[ ]; int l, int m, int r)
// merges a[1]...a[m] with a[m-1]...a[r]
{ int i, j;  //used for counters
     static Item aux[maxN};  //temporary array used for merge. Should be static since we don't
                             //want to reallocate this memory every time merge called.
     // Now we'll first copy original arrays into aux array and copy merged results into
     // original array
     for (i = m+1; i > l; i--) //copies right half in reverse order.
     //Now we merge results
     for (intk = l; k < = r; k ++)
     //used to figure out what to put in a[k]
     {if (aux[j] < aux[i] )
          //i starts at l by the way we set up for loop above
          // Also j = r to start
               a[k] = aux[j--];
          else a[k] = aux[i++]
     }
}
// If didn't have reverse order
```



if j array finished first need to have check to make sure didn't go out of array.

Idea of copying 2nd half on in reverse order called <u>bitonic</u> merging

Sorting algorithm

```
template <class Item>
void mergesort (Item a[ ], int l, int r)
```

```
    //sort a[l] ...a[r]
{   if (r < = l) return;
    int m = (r+l)/2; //calculate midpoint

    mergesort(a, l, m); //sort left half
    mergesort(a, m+1, r); //sort right half
    merge(a,l,m,r); //merge results
}
```

Hash tables

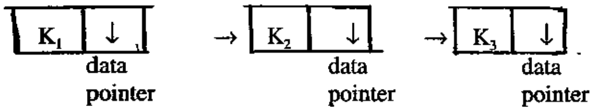
Often want to store data as key and associated information. This is called a dictionary. In "Real-life" dictionary, the key is a word and the definition is the associated date.

Other example,

key = Employee ID
associated date = work history

How to store things in a dictionary

(1) Use a flat file (unordered file)



To find an item have to search on average through half of list file.
To insert can just add to head to file so can be done in 0(1) time.

END OF LECTURE **

DECEMBER 6, 2000

Will go over Practice Final on Friday. Final Moore 100 Sunday 3-6

Last Day – finished up mergesort, talked about dictionaries/tables
Today – talk about hash tables

Dictionaries/Table

Key (employee ID)	Data (records)
999 25 9999	Ted Smith etc.
677 26 7777	Barb Walters etc.
⋮	⋮

Considered ways to store Tables

1. Flat file

Just a file/or linked list when data is written out in an unordered fashion. To find data given key expected search time O(N). To find data given key expected search time O(N). To add new entry can be done in O(1) time since can just keep as part of our structure a pointer to the last element in file or last element in list. To insert new item add after last item then update this pointer.

Can we implement tables so look up and insert both take O(1) time?
Yes.
1st way Open Addressing
Let N=total number of different possible keys. Make an array data[ ] of size N whose elements hold associated info.
Ex if k = 9 9 7 5 3 2 1 0 0 can look at data [9 9 7 5 3 2 1 0 0] to find info.
Insert can be just using assignment.
data[key] = info;
This is very wasteful. If want to store only 50 employees but use SSN as key need an array of size 10^9.
We'd like to achieve same result without wasting so much space.
Suppose we have N items to store. We'd like to use an array of size M not too much bigger than N and somehow map the keys into the set of size M.
This mapping is called a hash function.

h: K S → [M]
   e p   ↖
   y a
     c
     e   0, . . . M

To store data we'll use an array called hash table. To insert we'll usually do: table[h(key)]=info. To look up data for key K we'll usually look at the entry table[h(K)]. Since Key's usually of some fixed length not depending on our table size, we'll choose h(K) so that it takes constant time with respect to table size.

Ex Suppose want to store so employee using SSN as key

Could take hash function h(K) = sum of digits in SSN%10,003 then could use this to store employees in an array of size 10,003.

If store 1000 or 2000 employees amount of time to compute h(K) for a given key won't change.

Problem with above scheme: What if h(K) = h(K') but K≠K'?

This is called a collision.

One way to resolve this problem is to use linear probing.

Ex Suppose h(x)=x%7

0		
1	1	→ data for 1
2		
3		
4		
5		
6		

If try to store 8 get a collision

0	
1	1
2	8
3	
4	
5	
6	

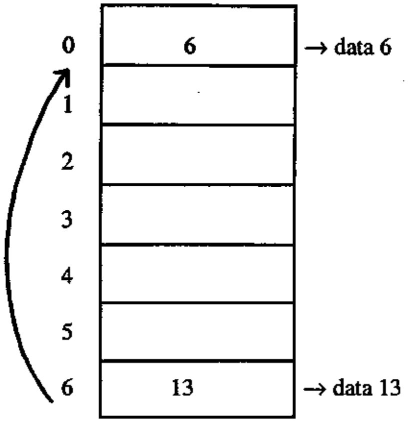
Store 8 and its data in 1st unoccupied space afterward.
To look up an entry with key K. Compute h(K) look if table [h(K)] has same key. If not step forward through table till find K.

Problem: Consider same h but table

0		
1		
2		
3		
4		
5		
6	13	→ data

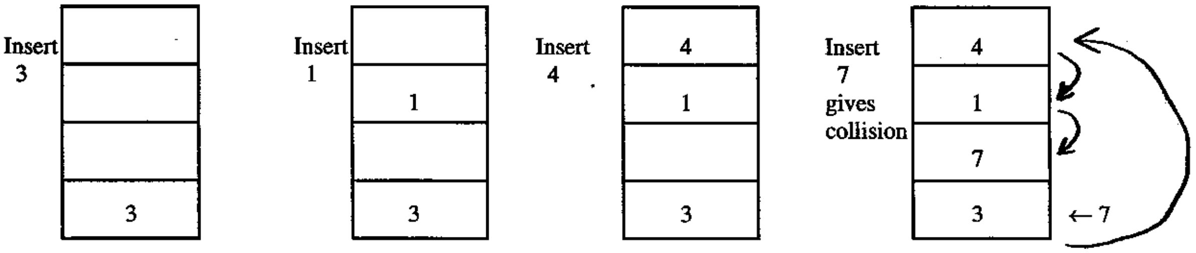
Want to store 6, data 6 so h(13)=h(6) so have a collision if step forward we go off end of array. Instead of stepping forward we step forward %7.

So would store 6 in location 0.



Provided not "too many" collisions hash tables give O(1) search/insert times.

Ex Suppose we insert 3147 into a hash table using x%4 and linear probing to resolve collisions. Draw resulting hash table.



END OF LECTURE**

Final has 7 problems. Each 5 pts. – 35 pts. total. No caps/cellphones close book/closed notes. Bring Photo ID

Practice Final

1) Define the following term

a) FILO – First In Last Out – Acronym used for describing inserting and removing objects from the stack.

b) tail recursion – a recursive function where there is one recursive call and it is the last statement in the function.

Ex int fact (int N)

```
    {    if (N == 0) return 1;
         return N * factorial (N-1);
    }
```

Point is that can replace this kind of recursion is can replace it easily without for loop.

c) memoization – also dynamic programming. The process of storing the results of subproblems of recursive pass as we do recursion so we don't have to recompute them.

Ex Fibonacci program did in class.

d) external node – in an M-ary ordered tree is a node without children (implementation with a NULL).

e) adaptive sorting – a sorting algorithm whose operations (the comparisons it makes) depends on the way the original unsorted data was sorted.

Ex 2nd insertion sort we did.

2) Write a program to reverse a list where Node's in list given as on Practice Final.

```
     template <class Item>
     Node <Item> * reverse (Node<Item> list)
// pointer to reversed list       // list to be reversed
     { Node<Item> *t, *y = list; *r=NULL;
```

1st // will store what's left to be reversed // reversed list so far

y → $\boxed{1}$ → $\boxed{2}$ → $\boxed{3}$

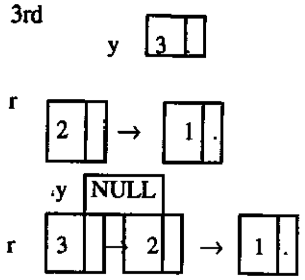
r → NULL

2nd

y → $\boxed{2}$ → $\boxed{3}$

r → $\boxed{1}$ ↓

3rd



done

```
      while (y != NULL)
      {  t = y → getNext();
            y → setNext(r);
            r = y;
            y = t;
      }
return r;
}
// end function
```

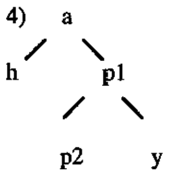
3) Briefly explain why the first version of insertion sort we described in class was nonadaptive and why it runs in time 0(N^2) to sort N elements.

```
      1st Version: template <class Item>
                  void sort(Item a[], int e, int r)
                     { for (int i = l + 1;  i <= r; i++)
                     for (int j = i; j > l; j--)
                           compexch (a [j-1] , a[j];
                     }
```

It's nonadaptive since will do same comparisons regardless of input array.
It's O(N^2) since outer for loop run N-1 times on array of size N.
Inner for loop does on average N/2 comparison exchanges. So run time

O[(N-1)N/2] = O(N^2)

4)



Write out nodes in

a) preorder
      ahp1p2y
b) inorder
   hap2p1y
c) postorder
      hp2yp1a
d) levelorder
      ahp1p2y

5) Write a program to compute # of external nodes in a tree

```
int exNodeCount (link h)
{ if (h == NULL) return 1;
      return exNodeCount ( h→ l) + exNodeCount(h→r);
}
```

END OF LECTURE AND SET # 10