Chapter 1 Highlights

--------------------

Emphasize importance of knowing the running time of an algorithm on

LARGE inputs (unless you have good reason to believe the algorithm

will not be run on such inputs). Even though computers keep getting

faster, data sets are getting even larger.

Math Review (or preview, if you are taking CSCE 222 now):

---------------------------------------------------

- rules of exponents

- rules of logarithms

- series formulas

- modular arithmetic

- proof by induction

- proof by contradiction

- recursion

- relationship between recursion and induction

We may come back to some of these topics on an as-needed basis as the

semester progresses: many of the students are taking CSCE 222 now.

C++ Review

----------remove

Classes are key organizing principle, as they promote modularity and

information hiding.

Note improved ways of doing initialization in later versions of C++.

Be sure to use preprocessor commands properly to make sure you don't try

to #include the same file multiple times when dealing with larger programs

consisting of multiple source fles.

Use vector and string classes instead of arrays and C-style strings.

The latter are bug magnets.

Make sure you replace the default destructor, copy/move constructor,

and copy/move assignment in your class if the default doesn't make

sense, especially likely if your class allocates memory dynamically

using "new" and you need deep copy instead of shallow copy.

Make your code templated within reason. There are many subtleties and

corner cases that are beyond the scope of this course.

Skip Section 1.7 on implementing 2-D matrices.

Useful exercises for practice (don't turn in):

1.5 (recursive function)

1.15 (make a Rectangle class, template findMax to refer to area and then to perimeter)

Chapter 2: Algorithm Analysis

------------------------------

An ALGORITHM is a sequence of unambiguous instructions or steps that solves

a problem.

A PROGRAM is the instantiation of an algorithm in a particular programming

language.

This chapter covers:

\* how to estimate the time taken by a program (or algorithm)

\* how to significantly reduce the running time

\* pitfalls with recursion

\* efficient algorithms for exponentiation and for GCD

Section 2.1: Math Background

-----------------------------

Algorithms are frequently described in PSEUDOCODE, intermediate between

natural language and some programming language. Why?

- more general than using a specific programming language

- avoids some tedious detail that could impede intuitive understanding

We want to estimate running times of algorithms.

Q: How to describe the running time?

A: As a function of the size of the input (denoted "n" or "N").

Q: But different inputs, even of the same length, might take different

amounts of time. What to do?

A: Focus primarily on WORST CASE. Why? Best case seems too optimistic,

average case is sometimes of interest but it can be hard to even define

what the average case input is.

Q: But running time depends on programming language, compiler, hardware,

level of multiprogramming, etc. What to do?

A: Abstract away from these details using ASYMPTOTIC ANALYSIS.

What is Asymptotic Analysis? Informally, measure time as a function

of input size and

\* ignore multiplicative constants and

\* ignore "lower order" terms

Focus on the behavior as the input size grows larger and larger.

Example: an algorithm whose time is "proportional" to the size of

the input is preferable to a competing algorithm whose time is "proportional"

to the square of the size of its input.

Why is this reasonable?

\* We always want faster programs that work on larger data sets

\* As input size increases, the effect of multiplicative constants

and lower order terms becomes negligible

\* Gives a common platform for comparing algorithms, independent of how

they might be implemented and executed.

The way we do asymptotic analysis is with BIG-OH NOTATION. It gives a

mathematically rigorous way to "ignore multiplicative constants and

lower order terms".

Here are the key definitions. T, f, g, h, and p are functions from

the natural numbers to the natural numbers.

Big-Oh: T(N) is O(f(N)) if there exist positive constants c and n\_0

such that T(N) <= c\*f(N) for all N >= n\_0.

Intuitively speaking, T(N) "<=" f(N), ignoring multiplicative constants

and lower order terms. We say f(N) is an "upper bound" on T(N).

\*\*\* draw plot \*\*\*

Big-Omega: T(N) is Omega(g(N)) if there exist positive constants c and n\_0

such that T(N) >= c\*g(N) for all N >= n\_0.

Intuitively speaking, T(N) ">=" g(N). We say g(N) is a "lower bound"

on T(N).

\*\*\* draw plot \*\*\*

Big-Theta: T(N) is Theta(h(N)) if T(N) is O(h(N)) and T(N) is Omega(h(N)).

Intuitively speaking, T(N) "=" h(N).

\*\*\* draw plot \*\*\*

little-oh: T(N) is o(p(N)) if for every positive constant c, there exists

n\_0 such that T(N) < c\*p(N) for all N > n\_0.

Intuitively speaking, T(N) "<" p(N).

\*\*\* draw plot \*\*\*

(There is also a little-omega but our textbook doesn't use it.)

Examples:

(i) Show 1000\*N is O(N^2).

Use c = 1 and n\_0 = 1000.

Could also use c = 100 and n\_0 = 10.

CSCE 222 goes more in-depth on this. For this class, we won't dwell

on the mathematical details of proving big-Oh, etc. Instead we will

rely on a repertoire of known results:

Rule 1: If T\_1(N) is O(f(N)) and T\_2(N) is O(g(N)),then

(a) T\_1(N) + T\_2(N) = O(f(N) + g(N)) which is the same as O(max(f(N),g(N))

(b) T\_1(N)\*T\_2(N) = O(f(N)\*g(N))

Rule 2: If T(N) is a polynomial of degree k, then T(N) = Theta(N^k).

Rule 3: log^k N is O(N) for any constant k. That is, logarithmic

functions, even if raised to a power, grow VERY SLOWLY.

Rule 4 (Figure 2.1 on p. 53): Asymptotically speaking, the following

functions are ordered like this (i.e., each one is little-oh of the next

one):

\* c ("constant")

\* log N ("logarithmic")

\* log^2 N ("log-squared")

\* N ("linear")

\* N log N

\* N^2 ("quadratic")

\* N^3 ("cubic")

\* 2^N ("exponential")

Section 2.2: Model of Computation

---------------------------------

We will use a somewhat simplified model of a real computer.

\* Instructions are executed sequentially

\* Each instruction takes exactly one time unit (e.g., addition,

multiplication, comparison, assignment) - be careful not to allow

"unreasonable" instructions

\* Fixed-size integers (usually)

\* Infinite memory

Section 2.4: Running-Time Calculations

---------------------------------------

Example:

input: array A of N integers

output: maximum element in A

1. currMax := A[0]

2. for i = 1 to N-1 do

3. if currMax < A[i] then

4. currMax := A[i]

5. return currMax

To calculate the running time:

Input size is N (assuming each integer is of unit size).

Step 1: one indexing operation, one assignment => O(1)

Step 2: This is shorthand for starting with i = 1, incrementing i by 1

and testing if result is at most N-1. Each increment and

test takes O(1) time. It is done N times.

Total time over all iterations is O(N).

The test fails the last time so the body of the for loop is done

N-1 times.

Step 3: One indexing, one comparison per iteration, N-1 iterations => O(N)

Step 4: One indexing, one assignment per iteration, N-1 iterations => O(N)

Step 5: O(1)

Total is O(1) + O(N) + O(N) + O(N) + O(1) = O(3N + 2) = O(N).

Note: there are some issues with the example given in Section 2.4.1

of the textbook. The example takes as input a single integer n and

computes the sum of the first n cubes (1^3 + 2^3 + ... + n^3), which

is shown to take O(n) time.

Issue #1: Earlier in the chapter, it was assumed that integers are of

fixed size. So n can never exceed some constant and thus the running

time is constant.

Issue #2: Even if we assume that integers can be of unbounded size,

the size of the input is going to be log\_2 n because all reasonable

computing devices uses binary to represent numbers. So as a function

of the input size, the running time is exponential, not linear.

Lesson: There are subtleties when analyzing algorithms that do numeric

computations, as to whether we want to measure performance based on the

magnitude (value) of the numbers or the size of their representations.

The example I just gave doesn't have these issues because we have N

different items as the input.

General rules for computing running time of algorithms:

-------------------------------------------------------

Rule 1, for loops: At most the time of all the statements inside

the for loop (including tests) TIMES the number of iterations.

Rule 2, nested loops: The total time of a statement inside a group

of nested loops is the running time of the statement TIMES the PRODUCT

of the sizes of all the nested loops.

Example:

for i := 0 to n-1 // body is done n times

for j := 0 to m-1 // body is done m times

k := k+1 // each iteration takes constant time

Total is O(1\*n\*m) = O(n\*m).

Rule 3, consecutive statements: ADD the times for all the statements.

As long as there is only a constant number of statements (as will be true

for any reasonable program, since programs are finite in size and don't

change depending on the input), this is the same as taking the MAXIMUM.

Example:

for i := 0 to n // this loop takes O(n) time

a[i] := 0

for i := 0 to n-1 // this loop takes O(n^2) time

for j := 0 to n-1

a[i] := a[i] + a[j] + i + j

Total is O(n + n^2) = O(n^2).

Rule 4, if/else: At most the time for the test plus the larger of

the times for the if clause and the else clause.

Recursive programs: Develop a recurrence relation and solve it (cf. CSCE 222).

Sections 2.3 and 2.4.3: Maximum Subsequence Sum Problem

--------------------------------------------------------

Given integers A\_1, A\_2, ..., A\_N, find the maximum value of Sum\_{k=i}^j A\_k.

(If all the integers are negative, define the answer as 0.)

Example: for input -2, 11, -4, 13, -5, -2, the answer is 20

(obtained from A\_2 + A\_3 + A\_4).

Algorithm 1: Brute force / exhaustive enumeration.

--------------------------------------------------

Suppose input is given in an array A.

Let i index the starting point, j index the ending point, and

add up all the numbers between indices i and j.

Let i start at index 0 and be increment up to n-1, and let j start at i

and increment up to n-1.

Keep track of the largest sum seen so far.

input: array A[0..n-1]

1. int maxSum := 0

2. for i := 0 to n-1

3. for j := i to n-1

4. thisSum := 0

5. for k := i to j

6. thisSum := thisSum + A[k]

7. if (thisSum > maxSum)

8. maxSum := thisSum

9. return maxSum

Time: Note that each for-loop test takes constant time (increment

the loop index and compare it to the limit).

step 1: O(1)

step 2: This for-loop test is done n+1 times => O(n)

The body of the for loop has n iterations

step 3: For each iteration of the enclosing loop, this for-loop test

is done AT MOST n+1 times (it starts at the current value of

i which keeps changing but is never less than 0).

Since the enclosing loop has n iterations => O(n^2)

The body of the for loop has AT MOST n iterations.

step 4: For each iteration, O(1) time. Number of iterations is

O(n\*n), since outer loop has n iterations and inner loop

has at most n iterations.

step 5: For each iteration of the enclosing inner loop, this for-loop

test is done AT MOST n+1 times (it starts at the current value of

i, which is at least 0, and ends at the current value of j, which

is at most n-1).

since the inner enclosing loop has at most n iterations and the

outer enclosing loop has n iterations => O(n^3)

step 6: Each execution takes constant time (one indexing, one addition,

one assignment). The number of times it is executed is

O(n^3) (inside three nested loops, each loop is done at most n

times) => O(n^3)

steps 7-8: Each execution takes constant time (one comparison and

maybe one assignment). The number of times they are executed

is O(n^2) (inside two nested loop, each loop is done at most n

times) => O(n^2)

step 9: O(1).

Grand total is the sum of the asymptotic times for all the lines,

which is the maximum, so O(n^3).

But maybe we over-counted! Maybe line 6 is actually executed significantly

fewer than O(n^3) times. More careful math, though, shows that it is not.

The exact number of times it is executed is

sum\_{i=0}^{n-1} sum\_{j=i}^{n-1} sum\_{k=i}^j 1 = (n^3 + 3n^2 + 2n)/6

which is Theta(n^3).

Hints for the algebra: Use these facts:

\* sum\_{i=1}^n i = n(n+1)/2

\* sum\_{i=1}^n (a+b) = sum\_{i=1}^n a + sum\_{i=1}^n b.

Algorithm 2:

------------

Algorithm 1 does unnecessary computation and we can get rid of the

inner for loop:

input: array A[0..n-1]

1. int maxSum := 0

2. for i := 0 to n-1

3. thisSum := 0

4. for j := i to n-1

5. thisSum := thisSum + A[j];

6. if (thisSum > maxSum)

7. maxSum := thisSum

8. return maxSum

<< Breakout rooms to convince each other of correctness and analyze the

running time. >>

Analysis of running time: It is easy to see that it is O(n^2) (that is,

big-oh, upper bound), since the outer for loop is done exactly n times and

the inner for loop is done AT MOST n times.

Let's do a more careful analysis, based on the fact that during the i-th

iteration of the outer for loop, the inner for loop is done (n-1)-i+1 = n-i

times:

Sum\_{i=0}^{n-1} Sum\_{j=i}^{n-1} 1 = Sum\_{i=0}^{n-1} (n-i)

= Sum\_{i=0}^{n-1} n - Sum\_{i=0}^{n-1} i

= n^2 - (n-1)\*n/2

= (n^2)/2 + n/2, which is Theta(n^2).

Algorithm 2:

------------

Algorithm 1 does unnecessary computation and we can get rid of the

inner for loop:

input: array A[0..n-1]

1. int maxSum := 0

2. for i := 0 to n-1

3. thisSum := 0

4. for j := i to n-1

5. thisSum := thisSum + A[j];

6. if (thisSum > maxSum)

7. maxSum := thisSum

8. return maxSum

<< Breakout rooms to convince each other of correctness and analyze the

running time. >>

Analysis of running time: It is easy to see that it is O(n^2) (that is,

big-oh, upper bound), since the outer for loop is done exactly n times and

the inner for loop is done AT MOST n times.

Let's do a more careful analysis, based on the fact that during the i-th

iteration of the outer for loop, the inner for loop is done (n-1)-i+1 = n-i

times:

Sum\_{i=0}^{n-1} Sum\_{j=i}^{n-1} 1 = Sum\_{i=0}^{n-1} (n-i)

= Sum\_{i=0}^{n-1} n - Sum\_{i=0}^{n-1} i

= n^2 - (n-1)\*n/2

= (n^2)/2 + n/2, which is Theta(n^2).

\*\*\*Note: You can skip Algorithms 3 and 4 given next\*\*\*

Algorithm 3:

------------

Using the power of recursion, we can reduce the running time even more!

We will use an algorithm design paradigm called "divide and conquer"

(we'll see more algorithms like this later and even more are covered in

CSCE 411).

VERY high-level pseudocode:

Recursive algorithm operates on a contiguous subsequence of array A,

from index l (for left) to index r (for right). Top level call has

l = 0 and r = n-1 (i.e., the entire array.

Base case is when l = r, so we are just dealing with a single element of A.

Return the element if it is positive and otherwise.

Inductive case is when l < r, so we are dealing with more than one element

of A.

Divide: Call the algorithm recursively in the left half of the current

segment of A and on the right half.

Can we just choose whichever answer is larger?

No, because there might be a larger sum due to a subsequence that

crosses the center boundary, so we have to calculate those as well.

It turns out that the running for this algorithm is O(n log n) (calculated

using techniques that you will learn in CSCE 222 for solving recurrences).

Algorithm 4:

-----------

input: array A[0..n-1]

1. int maxSum := 0

2. thisSum := 0

3. for j := 0 to n-1

4. thisSum := thisSum + A[j]

5. if (thisSum > maxSum)

6. maxSum := thisSum

7. else if (thisSum < 0)

8. thisSum := 0;

9. return maxSum

Correctness is not obvious (and ideally should be proved rigorously).

Like Algorithm 2, j is keeping track of the end of the current sequence

and we have dispensed with the use of i (for the start of the current

sequence).

A negative subsequence cannot possibly be a prefix of the optimal

subsequence, so we just drop its contribution in lines 7-8.

Running time can easily be seen to be O(n), which is the best possible

since every input number needs to be considered.

Section 2.4.4: Algorithms with Logarithmic Running Time

--------------------------------------------------------

This sounds odd at first since the entire input needs to be read into

the program and that will take linear time.

However, you might be using an algorithm A as a subroutine in a larger

algorithm B which has already read in the input. So we can just count

the time taken by A.

Classic example is binary search:

--------------------------------

input: integer x and array A of SORTED integers, already in memory

output: index i such that A[i] = x; if x does not appear in A then -1

1. low := 0

2. high := n-1

3. while (low <= high)

4. mid := (low + high)/2

5. if (A[mid] < x)

6. low := mid + 1

7. else if (A[mid] > x)

8. high := mid - 1

9. else

10. return mid // found

endwhile

11. return -1 // not found

Correctness: Because A is initially sorted, we'll eventually find the

place where x should be.

Running time: Each iteration of the while loop takes constant time.

How many iterations of the while loop? In the worst case, x doesn't

appear in A and we keep going until low > high. Each iteration cuts in

half the size of the array segment being considered:

n, n/2, n/4, n/8, ..., 1

This takes log\_2 n iterations.

So total time is O(log n).

Another example is Euclid's GCD algorithm.

-----------------------------------------

The greatest common divisor (gcd) of two integers is the largest integer

that divides both of them. Euclid was a Greek mathematician who lived

in the 300's BC.

Example: gcd(50,15) = 5.

input: two integers m and n, m >= n

output: gcd(m,n)

1. while (n != 0)

2. rem := m % n // remainder, or mod

3. m := n

4. n := rem

endwhile

5. return m

Algorithm continually computes remainders until reaching 0.

Example: m = 1,989 and n = 1,590.

rem := 1989 % 1590 = 399

m := 1590

n := 399

rem := 1590 % 399 = 393

m := 399

n := 393

rem := 399 % 393 = 6

m := 393

n := 6

rem := 393 % 6 = 3

m := 6

n := 3

rem := 6 % 3 = 0

m := 3

n := 0

n = 0 so return m = 3

Correctness relies on some number theory properties.

Running time:

Theorem 2.1: If m > n, then m mod n < m/2.

Proof:

Suppose n < m/2. Since the remainder must be less than n,

it is also < m/2.

Suppose n > m/2. Then n goes into m once with a remainder of m - n,

which is less than m/2.

QED

The theorem implies that after TWO iterations of Euclid's algorithm,

the remainder is at most half what it used to be.

Why? Let m\_i and n\_i be the values of m and n during iteration i.

Then m\_{i+1} = n\_i and n\_{i+1} = m\_i mod n\_i.

And finally m\_{i+2} = n\_{i+1} = m\_i mod n\_i.

Once the remainder gets to 0, we are done. So the number of iterations

required is at most 2\*(log n) = O(log n).

This is logarithmic in the MAGNITUDE (or value) of the smaller argument,

but this would be linear in the representation, assuming binary.

Third example: Exponentiation

------------------------------

We want an algorithm to raise an integer x to an (nonnegative) integer power n.

Assume memory locations are large enough to hold big numbers.

Count the number of multiplications.

Idea 1: compute x\*x\*x...\*x - uses n-1 multiplications

Idea 2: use recursion! If we already have x^{n/2}, we can get

x^n from x^{n/2} \* x^{n/2}, which is one multiplication.

power(x,n):

if (n = 0) // base case

return 1

// recursive cases follow

if (n is even)

return power(x\*x,n/2) // returns (x^2)^{n/2}, which equals x^n

else // n is odd

return power(x,n-1)\*x // returns x^{n-1}\*x, which equals x^n

Note that each recursive call gets closer to the base case (second argument

is either cut in half or decremented by 1).

Example execution:

Suppose x = 2 and n = 21. Then the sequence of recursive calls is:

/ power(2, 21)

| / power(2, 20)

| | / power(4, 10)

| | | / power(16, 5)

| | | | / power(16, 4)

| | | | | / power(256, 2)

| | | | | | / power(2^16, 1)

| | | | | | | / power(2^16, 0)

| | | | | | | \ returns 1 // base case

| | | | | | \ returns 1 \* 2^16 = 2^16

| | | | | \ returns 2^16

| | | | \ returns 2^16

| | | \ returns 2^16 \* 16 = 2^20

| | \ returns 2^20

| \ returns 2^20

\ returns 2^20 \* 2 = 2^21

Correctness: Should be clear from rules of exponents.

Running time: Keep making recursive calls until second argument is 0.

Second argument, the exponent, is either cut in half (if even) or

reduced by 1 (if odd). If it is odd, then the next time it is even

and therefore cut in half. Thus it takes at most 2 iterations for the

exponent to be cut in half, so it takes at most 2\*(log\_2 n) iterations,

which is O(log n) iterations. Each iteration does one multiplication,

so the total number of multiplications is O(log n).

Again, we have the funny business with how the running time relates

to the size of the input. Here, n is the MAGNITUDE of the exponent

in the input. If we represent the input in binary, then the running

time is linear instead of logarithmic.

Section 2.4.5: Limitations of Worst-Case Analysis

--------------------------------------------------

Sometimes actual experiments show better running time than the analysis.

Why?

(1) The analysis is too pessimistic and might be tightened.

(2) The inputs on which the experiments are run do not include those

inputs that elicit the worst-case behavior of the algorithm.

And if you don't care (or have) large inputs, asymptotic analysis is

not the right tool. It may be that an algorithm doesn't exhibit its

good behavior until inputs get large. Also, if a very efficient

algorithm is complicated and error-prone to implement, it might not

be worth it for small problem sizes.

Chapter 3: Lists, Stacks and Queues

------------------------------------

Section 3.1: Abstract Data Types (ADTs)

----------------------------------------

Let's look at some examples before giving a more precise definition.

Example 1: A variable, whose state is a value over some set of values,

and whose operations are read, which returns the current value of the state,

and write, which changes the value of the state to its argument.

Correct:

[12] read(12) [12] write(8) [8] write(5) [5] read(5) [5] read(5) [5]

Incorrect:

[12] read(12) [12] write(8) [8] read(12) [8]

Example 2: A (mathematical) set, whose state is a set of integers

and whose operations are insert(x), which adds x to the set, and remove(x)

which removes x if it is present and returns an indication as to whether

it succeeded.

Correct:

{1,2} insert(3) {1,2,3} insert(4) {1,2,3,4} remove(1)-yes {2,3,4} remove(5)-no {2,3,4}

Incorrect:

{1,2} insert(3) {1,2,3} remove(1)-yes {2,3} remove(1)-yes {2,3}

The state of the variable is just a single value (which fits will with

a programming language variable).

For the state of the set, I used math notation, which would need to be

represented with something less abstract, and more concrete, in a program.

In fact, we don't even need the abstract states, although they are

useful for intuition. Given an assumed initial state, we only need

the sequence of operations, as the intermediate states can be deduced.

Example 1: Assuming initial state is 12,

Correct:

read(12) write(8) write(5) read(5) read(5)

Incorrect:

read(12) write(8) read(12)

Example 2: Assuming initial state is {1,2},

Correct:

insert(3) insert(4) remove(1)-yes remove(5)-no

Incorrect:

insert(3) remove(1)-yes remove(1)-yes

Definition of ADT: A set of states, a set of operations, and a set

of correct sequences of operations (that capture the desired behavior

of the object).

KEY POINT: The specification of an ADT says NOTHING about how the

state is represented and how the operations are implemented. All it

cares about is the interface, the interaction between the user, who

invokes operations, and the object, which provides responses to the

operation invocations.

The C++ notion of a class matches well with the notion of an ADT as

a class provides state (data members) and operations (function members),

provides a interface to the user (public), and can hide the implementation

(private).

Ideally, the implementation of an ADT is completely hidden so that it

can be changed with no effect on the correctness of the code that uses

objects of the ADT. (Performance might be affected.)

Section 3.2: The List ADT

-------------------------

Specification:

A state is a sequence of values, A\_0, A\_1, A\_2, ..., A\_{N-1}.

Typical operations are

- printAll: output all the elements of the list in order

- findVal(x): returns position in list (i.e., subscript) of first occurrence

of value x

- findPos(k): returns value in the k-th position in the list

- insertFront(x): adds x to the beginning of the list

- insertBack(x): adds x to the end of the list

- removeFront: removes and returns element at front of the list

- removeBack: removes and returns element at back of the list

etc. You can have variations on these, or others, and you should

also specify what happens in corner cases, say if the list is empty.

The correct sequences of operations should be intuitively clear from

the names of the operations. (It's more of a pain, with little gain,

to give a formal specification.)

Section 3.2.1: List Implementation #1: Array

----------------------------------------------

State: Store the sequence of values consecutively in the array,

in indexes 0 through n-1, where n is the number of values currently

in the list.

Typical operations and their asymptotic worst-case running times, letting

n be the number of values in the array:

- printAll: loop through the array in order, outputting the values.

O(n)

- findVal(x): loop through the array in order until finding x

O(n)

- findPos(k): returns value in the k-th element of the array

O(1) \*\*\* arrays support constant-time indexing

- insertFront(x): shift down all the array elements to make room

for x at the beginning

O(n)

- insertBack(x): adds x to the end of the array

O(1) as long as there is enough space remaining in the array,

otherwise allocate a larger array and copy elements from

old array to new array, leading to O(n) time

- removeFront: shift up all the array elements

O(n)

- removeBack: return element at index n-1 and decrement n

O(1)

Section 3.2.2: List Implementation #2: Linked List

----------------------------------------------------

In a linked list, each value is stored together with a pointer

that contains the address of the next value in the List.

The data is allocated dynamically on the heap.

Typical operations and their asymptotic worst-case running times, letting

n be the number of values in the List:

- printAll: traverse through the linked list in order, outputting

the values. O(n).

- findVal(x): traverse through the linked list in order until finding x

O(n)

- findPos(k): traverse through the linked list until reaching the k-th

element.

O(n) \*\* worse than array! \*\*

- insertFront(x): Assuming a pointer to the head of the list is kept,

make a new list element, have it point to the old head, and update

head pointer to point to new element.

O(1) \*\* better than array! \*\*

- insertBack(x): Assuming a pointer to the tail of the list is kept,

make a new list element, have tail element point to it, and update

tail pointer to point to new element.

O(1)

- removeFront: Change head pointer to point to what head element

pointed to (and possibly delete removed element from memory)

O(1) \*\* better than array! \*\*

- removeBack: Get access to 2nd-to-last element in the list, have it

point to nothing, have tail pointer point to 2nd-to-last element.

(possibly delete removed element from memory)

O(n) if singly-linked list because need to traverse entire list to

find 2nd-to-last element

O(1) if doubly-linked list (so each list element points to previous

element as well as next element)

//////////////////////

// SUMMARY:

// array-based lists are faster at direct access (indexing)

// linked-list based lists are faster at inserting and removing

// elements from arbitrary positions in the list

// array-based lists require extra space to avoid frequent

// copying of the array when it runs out of space

// linked-list based lists require extra space for the pointers

/////////////////////

Sections 3.3-3.5: List ADT implementations in C++

--------------------------------------------------

The List (with a capital L) ADT is implemented in C++ most frequently

with either the vector container or the list (with a lower case ell)

container.

C++ container "vector" is an improved version of an array that supports

- resizing

- bounds-checking for indexing

C++ container "list" (lower case ell) is implemented with a doubly-linked

list.

Section 3.3 reviews the interfaces for C++ vector and list containers,

and reviews iterators, which are crucial for using containers.

Remember that an iterator is an abstraction of a pointer.

Section 3.4 goes over how to implement vector using built-in array.

Main idea is that initially an array is allocated on the heap of a

certain size, called the capacity. Whenever the number of elements

stored in the array (called the size) exceeds the capacity, a new

array whose capacity is (usually) double the capacity of the current

array is allocated and all the data from the old array is copied

into the new array. This is time-consuming but it can be shown that

on average, it only takes constant time -- the time for the expansion

is "amortized" over the time for all the operations that don't require

expansion.

Section 3.5 goes over how to implement a doubly-linked list in C++.

Here are some of the key ideas.

\* Each element of the list is a Node object, which contains

- data

- pointer to previous node

- pointer to next node

--------------------------

class Node {

public:

// data members

int data;

Node\* prev;

Node\* next;

// constructor with default parameters

Node(int i = 0, Node\* p = nullptr, Node\* n = nullptr) :

data{i},prev{p},next{n} {}

};

--------------------------

\* the list will have a dummy "sentinel" node at the beginning and another

dummy "sentinel" node at the end. This is not absolutely necessary,

but can simplify some of the code because there are fewer special case

that can arise if the (abstract) List is empty or only has one element.

\* the list data members are

- the size of the list (number of data items stored in it)

- head, pointer to beginning sentinel node

- tail, pointer to final sentinel node

--------------------------

class List {

public:

int sz;

Node\* head;

Node\* tail;

List() {

sz = 0;

head = new Node;

tail = new Node;

head->next = tail;

tail->prev = head;

}

// ...

};

--------------------------

\* code fragment for inserting:

--------------------------

// insert a new node containing x immediately BEFORE \*cur

void insert(Node\* cur, int x) {

// make a new node for the new data value that goes in between

// \*cur and \*cur's predecessor

Node\* newNode = new Node{x,cur->prev,cur};

// adjust next pointer for \*cur's original predecessor

cur->prev->next = newNode;

// adjust prev pointer for \*cur

cur->prev = newNode;

sz++;

}

--------------------------

\* code fragment for erasing:

--------------------------

// erase the node \*cur, return pointer to item AFTER the erased element

Node\* erase(Node\* cur) {

// get return value

Node\* retVal = cur->next;

cur->prev->next = cur->next;

cur->next->prev = cur->prev;

delete cur;

sz--;

return retVal;

}

--------------------------

Improvements given in the code in the textbook:

\* data items can be any of any type, not just ints: accomplished

using templates

\* copy and move semantics are taken care of. Copy issue: Suppose

you want to make a copy of a List object.

List myList1{};

// ... code to put some data in the list

List myList2{};

myList2 = myList1;

Only the "bookkeeping" information of myList1 has been copied,

namely, sz, head, and tail. The nodes are shared! Probably not

what you want, and very dangerous.

\* There is no encapsulation. The user code has direct access to the

pointers into the memory of the List object and thus it can wreak

havoc on it (turn the pointers into spaghetti, etc.).

One way around that is to use iterators to abstract away from the

pointers and control the access. Dereferencing an iterator only

gives access to the data, but not to the prev and next pointers.

The only way to change the linked structure of the list is through

functions such as insert and erase.

\* Error-checking (discussed but not fully implemented).

Section 3.6: Stack ADT

-----------------------

Intuition:

- a List in which elements can be added to the end only and removed from

the end only

- thus every time you remove an element you get the one that was added

most recently: last-in-first-out (LIFO)

Specification:

--------------

\* state: a sequence of elements

\* (typical) operations:

- push(x): add x to the end of the sequence

- pop(): remove and return the element at the end of the sequence

- top(): return without removing the element at the end of the sequence

- size: number of elements in the sequence

If you try to do pop or top on an empty Stack, some kind of error message

should be returned.

<< draw picture >>

Implementations:

---------------

#1: Use a singly linked list.

- push: insert at front of list

- pop: remove first element in list

- top: return value in first element in list

- size: keep track of size when pushing and popping and return its value,

All the operations are constant time.

#2: Use a vector (or an array) together with the index of the last

element added to the stack (initially -1), which is conceptually the

top of the stack.

- push: push\_back on the vector, increment the top index

- pop: pop\_back on the vector, decrement the top index

- top: return value in the vector at the top index

- size: return top index (plus 1)

All the operations are constant time. (The usual caveat about the vector

needing to resize its array.)

Applications:

-------------

#1: checking for syntax errors related to balancing parentheses

(and brackets, and curly braces, etc.) in a program (or any kind

of text file)

make an empty stack S

for each c in the file

if c is an opening symbol then push c onto S endif

if c is a closing symbol then

pop S into c'

if c and c' do NOT match then ERROR

endif

endfor

if S is not empty then ERROR

.......................................

#2: evaluating postfix arithmetic expressions

Postfix means that the operator (operation) appears AFTER the two operands

(data):

x y + instead of x + y

The advantage of postfix notation is that you do not need to use

parentheses to change the default order of precedence of operations:

infix: (x + y)\*z (which is not the same as x + y\*z)

postfix: x y + z \* (compare to x y z \* +)

Algorithm:

make an empty stack S

while (input not empty)

get next input token t

if t is an operand then S.push(t)

if t is an operator then

x := S.pop()

y := S.pop()

z := apply t to x and y

S.push(z)

endif

endwhile

return S.pop()

Example: 6 5 2 3 + 8 \* + 3 + \*

Running time: Theta(n), where n is the number of tokens in the input.

#3: Converting infix expression to postfix expression

Assume we only have operators + and \* and that the input infix expression

is properly formatted.

Algorithm:

make an empty stack S

while (input not empty)

get next input token t

if t is an operand then print it

if t is '(' then S.push(t)

if t is ')' then

pop and print everything on S until reaching the matching '('

get rid of the '(' without printing it

if t is '\*' then pop and print everything on S until reaching '+' or '('

(or stack is empty) and then push '\*'

if t is '+' then pop and print everything on S until reaching '('

(or stack is empty) and then push '+'

endwhile

pop and print everything remaining on S

Example: Input is: a + b\*c + (d\*e + f)\*g

Output will be: a b c \* + d e \* f + g \* +

.......................................

#4: Runtime System Handling of Function Calls

When a program is executing, the system needs to store some

information for each function call (plus main), notably all the local

variables declared in that function, plus other bookkeeping

information including return addresses. This packet of information is

called a STACK FRAME or ACTIVATION RECORD.

Stack frames are organized like a stack: when function A calls function B,

the stack frame for B is pushed onto the run-time stack (on top of A's

stack frame). When function B returns, its stack frame is popped off

the run-time stack.

Warning: If you are careless in your programming, you can run out of

stack space (stack overflow). One way that can happen is with a function

that exhibits "tail recursion":

void bad\_print(vector<int> V, int i) {

if (i == V.size()) return; // base case

cout << V[i] << " ";

bad\_print(V,i+1);

}

If you call this on a vector of size one million, the system will try to

make a million stack frames (plus, since V is passed by value, there

will be a million copies of V in the heap).

Section 3.7: The Queue ADT

---------------------------

Intuition:

- a List in which elements can be added only to one end and removed

only from the other end

- thus every time you remove an element you get the one that was added

least recently: first-in-first-out (FIFO)

Specification:

--------------

\* state: a sequence of elements

\* (typical) operations:

- enqueue(x): add x to the end of the sequence

- dequeue(): remove and return the element at the beginning of the sequence

- peek(): return without removing the element at the beginning of the

sequence

- size: number of elements in the sequence

If you try to do dequeue or peek on an empty Queue, some kind of error message

should be returned.

<< draw picture >>

Implementations:

---------------

#1: Use a linked list. Simple and results in O(1) time operations.

#2: Use an array (or vector).

- To avoid shifting the data values back and forth in the array when

enqueues and dequeues are done, keep two indices into the array

indicating the location of the head and tail.

- To avoid running off the end of the array if, say one were to keep

doing enqueues, wrap around to the front of the array.

Called "circular array" implementation.

<< draw example on p. 114 >>

Applications:

-------------

#1: used in some fundamental graph theory algorithms, which we'll

see later in the semester

#2: job queues to access services such as printers and file servers

#3: an output stream (cf. output of the infix-to-postfix conversion

algorithm)

Chapter 4: Trees

-----------------

Frequently we need a data structure whose state is a collection of items

with operations to insert an item, remove an item, and search for an item.

(We will formalize this as a "Dictionary" ADT later on.)

Our options so far for implementation are (using O as a sloppy shorthand

for Theta):

- linked list:

- search: O(n)

- insert: O(1) (assuming you are at the insertion point)

- remove: O(1) (assuming you are at the removal point)

- (unsorted) array:

- search: O(n)

- insert: O(1) (put at the end)

- remove: O(n) (close up the gap)

- (sorted) array:

- search: O(log n) (use binary search)

- insert: O(n) (to keep array sorted)

- remove: O(n) (close up the gap)

None of these options is great. Can we find a data structure that

has the best of all worlds? Or close to it?

Yes! In fact, there are several, based on the notion of "tree" that

have O(log n) time for all these operations, either on average or even

in worst case.

Section 4.1: Tree Preliminaries

--------------------------------

Definition: A TREE is defined by two sets,

- V, the set of "nodes" (or "vertices") and

- E, the set of "edges", where each edge is an ordered pair of

nodes (u,v), where u is the "child" of v, and v is the "parent" of u

Furthermore,

- there is a distinguished node called the "root"

- every node except the root has exactly one parent and the root has no parent

Draw trees with the root at the top, oddly enough.

Terminology

- a node with no children is a "leaf"

- nodes with same parent are "siblings"

- a "path" from node n\_1 to node n\_k is a sequence of nodes

n\_1, n\_2, n\_3, ..., n\_k such that each n\_i is the parent of n\_{i+1}

(in this definition, paths go down the tree, away from the root, but we can

also define paths as going up the tree, toward the root)

- the "length" of a path is the number of EDGES on the path

- the "depth" of node n\_i is the length of the (unique) path from the

root to n\_i

- the "height" of node n\_i is the length of the LONGEST path from n\_i

to a leaf

- the "depth" of a tree is the maximum depth of any node in the tree

- the "height" of a tree is the height of the root

Don't confuse depth/height of a specific node with that of the entire tree!

Facts:

\* A tree with n nodes has n-1 edges.

\* The depth of a tree equals its height.

A tree can be implemented with a linked structure, where the parent

has pointers to all its children:

class TreeNode {

Object element; // keep some data associated with the node

vector<TreeNode\*> children; // pointers to the child nodes

};

However, there is some space wastage with this if the number of children

varies wildly from node to node. An alternative popular approach is

to keep all the children of a node in a linked list: the parent only

has a pointer to the beginning of its child list, and each node has a

pointer to its next sibling:

class TreeNode {

Object element;

TreeNode\* firstChild;

TreeNode\* nextSibling;

};

< Draw Figures 4.2 and 4.4, p. 122-123 >

Trees model many situations in applications where there is some kind

of hierarchical structure.

- employee organization charts

- operating system file structures

- ...

A common operation on a tree is to visit each node exactly once,

called a "traversal". During the visit, an application might do

some computation, for instance print out the data associated with

that node. There are three kinds of traversals:

PREORDER TRAVERSAL: ("pre" because we visit each node BEFORE

recursively traversing the subtrees rooted at the node)

Algorithm preorder(T,p):

input: tree T and node p // in top level call, p is the root of T

output: whatever the visits do

perform the visit action for node p

for each child q of p do

preorder(T,q)

< draw example of preorder traversal on tree in Fig 4.2 >

Application: Printing out the names of all the files and subdirectories

in a file system, with indentation to indicate structure.

What is the running time of a preorder traversal?

- There is exactly one recursive call for each node in the tree.

- The nonrecursive work done in each call takes O(1) time

(assuming the visit action is constant time).

- So the total is O(n), where n is the number of nodes in the tree.

POSTORDER TRAVERSAL: ("post" because we visit each node AFTER

recursively traversing the subtrees rooted at the node)

Algorithm postorder(T,p):

input: tree T and node p // in top level call, p is the root of T

output: whatever the visits do

for each child q of p do

postorder(T,q)

perform the visit action for node p

< draw example of postorder traversal on tree in Fig 4.2 >

Same running time analysis as for preorder traversal.

Application: Calculating the total amount of disk space used by a directory

in a file system, and all its subdirectories. Can we use preorder

traversal for this application?

Section 4.2: Binary Trees

-------------------------

Definition: A "binary tree" is a special case of a tree in which every

node has AT MOST 2 children. The children of a node are oriented in

the sense that one is considered the left child and the other the

right child. Either one, or both, might be missing.

< draw example >

INORDER TRAVERSAL: Since this is a special case of tree, we can of

course apply pre-order and post-order traversals to it. But we can

also get another kind of traversal, called "in-order", that doesn't

make sense for non-binary trees:

Algorithm inorder(T,p):

input: tree T and node p // in top level call, p is the root of T

output: whatever the visits do

inorder(T,left(p)) // left(p) is the left child of p

perform the visit action for node p

inorder(T,right(p)) // right(p) is the right child of p

Same running time analysis as for preorder and postorder traversals.

Implementation: Since there are never more than two children, we can

go with the conceptually simpler approach of putting two pointers in

each TreeNode, one to the left child and one to the right child.

class BinaryNode {

Object element; // the data in the node

BinaryNode\* left; // pointer to left child

BinaryNode\* right; // pointer to right child

};

Application: Use a binary tree to represent an arithmetic expression.

The leaves are the operands and the internal nodes (non-leaves) are

the binary operators (e.g., \* and +).

Example: expression tree for (a+b\*c)+((d\*e+f)\*g)

< draw tree in Fig 4.14 on p. 129 >

Note that if you do an inorder traversal of an expression tree and enclose each

recursive call in parentheses, you get an (overly-parenthesized) infix expression.

Algorithm to convert a postfix expression into an expression tree:

make an empty stack S

while (input is not empty)

let t be the next token in the input

if t is an operand then

create a one-node tree, pointed to by T, in which the node's data is t

S.push(T)

if t is an operator then

T1 := S.pop()

T2 := S.pop()

create a tree, pointed to by T, s.t.

- the root's data is t

- the root's left child pointer is T2

- the root's right child pointer is T1

S.push(T)

endwhile

Example execution:

Input: ab+cde+\*\*

Output:

\*

/ \

+ \*

/ \ / \

a b c +

/ \

d e

If you want to check your understanding:

Using the tree in Fig. 4.74 (p. 183):

- determine height and depth of each node

- determine height and depth of the entire tree

- give the result of preorder, inorder, and postorder traversals of the tree

Section 4.3: Binary Search Trees

---------------------------------

Definition: A binary tree is a BINARY SEARCH TREE (BST) if the data items

associated with the nodes of the tree are drawn from a totally ordered

set (i.e., a set on whose elements a "less than" operation is defined)

and for every node X in the tree,

(1) all the data items in the subtree rooted at the left child of X are

less than the data item in X, and

(2) all the data items in the subtree rooted at the right child of X are

greater than the data item in X.

Examples:

6 This is a binary search tree

/ \ (check the property for all the nodes)

2 8

/ \

1 4

/

3

6 This is not binary search tree

/ \ (cf. 7 is greater than 6)

2 8

/ \

1 4

/ \

3 7

Typical operations on a binary search tree T:

\* contains(x) (also known as "search")

\* findMin()

\* findMax()

\* insert(x)

\* remove(x)

\* isEmpty()

\* printTree()

Implementation: Use the binary tree linked structure from earlier

with a data member "root" that points to the root node.

For simplicity, assume the data items are ints.

\* contains(x):

--------------

bool contains(int x) const // public interface of the BST class

{

return contains(x,root); // calls a private function, starting at root

}

Here's the recursive implementation of the private function:

bool contains(int x, BinaryNode\* t) const

{

if (t == nullptr)

return false; // x not found

else if (x < t->element)

return contains(x,t->left); // continue search in left subtree

else if (x > t->element)

return contains(x,t->right); // continue search in right subtree

else

return true; // found!

}

Example: Searches in the BST drawn above.

Worst-case running time: Theta(n), where n is the number of nodes in the

tree. This occurs if the tree is a chain (every node has one child,

except for the unique leaf).

This is tail recursion and can be very inefficient (lots of stack frames)

if n is large, but it's easy to rewrite this with a while loop instead

of recursion.

\* findMin and findMax:

----------------------

The minimum element in a BST is found by following left children, starting

at the root, until no left child exists.

Similarly, the maximum element is found by following right children,

starting at the root, until no right child exists.

Examples: Search for min and max in BST drawn above.

int findMin() const // public interface of the BST classs

{

// calls a private function, starting at the root

BinaryNode\* p = findMin(root);

// return the data stored in the node found by the private findMin

return p->element;

}

Here's the recursive implementation of the private function:

BinaryNode\* findMin(BinaryNode\* t) const

{

if (t == nullptr) // empty tree

return nullptr;

if (t->left == nullptr) // no left child

return t;

return findMin(t->left);

}

Worst-case running time is Theta(n), which occurs if the tree is a chain

always going to the left.

------------

findMax is analogous. Here we'll show the private function implemented

using a while loop instead of recursion:

BinaryNode\* findMax(BinaryNode\* t) const

{

if (t != nullptr) { // check for non-empty tree

while(t->right != nullptr) { // keep going right as long as possible

t = t->right;

}

}

return t;

}

----------------

\* insert:

---------

For simplicity, assume we don't want to have duplicates in the tree.

To insert x, first search for x as above, and if it is found, then do

nothing.

If x is not found, then we have found the place where x should go.

Example: Insert 5 into BST drawn above.

void insert(int x) // public interface of the BST class

{

insert(x,root); // calls a private function, starting at root

}

Here's the recursive implementation of the private function.

Note that the second argument is a pointer that is passed by reference!

void insert(int x, BinaryNode\* & t) // private function

{

if (t == nullptr) // found place for x, make new node

t = new BinaryNode{x,nullptr,nullptr};

else if (x < t->element) // continue search for insertion point

insert(x,t->left); // by going to the left

else if (x > t->element) // continue search for insertion point

insert(x,t->right); // by going to the right

else

; // x is already in the tree, do nothing

}

Question: Why is the new node linked in properly to its parent?

Answer: Because t is of type BinaryNode\* &.

Look at this example (and see powerpoint slides).

- Tree has root 6 with left child 4, which has left child 3.

- Call to (public) insert(5) makes a stack frame with memory location

named x that holds 5.

- Call to (private) insert makes a stack frame with memory location

named x (but we'll call it x' to disambiguate) that holds 5 and a

variable named t (but we'll call it t' to disambiguate) that is not

a separate memory location but is a synonym for the memory location

named root (which holds the address of a BinaryNode object).

- Call to (private) insert makes a stack frame with memory location

named x (but we'll call it x'') that holds 5 and a variable named t

(but we'll call it t'') that is not a separate memory location but is

a synonym for the memory location t'->left.

- Call to (private) insert makes a stack frame with memory location

named x (but we'll call it x''') that holds 5 and a variable named t

(but we'll call it t''') that is not a separate memory location but

is a synonym for the memory location t''->right.

- When the new tree node is made with new, its address is assigned to t''',

which means its address is assigned to t''->right, which is the same as

root->left->right. So the new node is properly attached to the tree.

Worst-case running time is Theta(n), which occurs if the tree is a chain

and x goes at the end of the chain.

----------------

\* remove:

---------

First, search for x as above. If it is not found, then do nothing.

If it is found, we want to remove that node from the tree.

But we can't mess up the tree structure!

Case 1: x is in a leaf node. Then just remove it (and set the appropriate

child pointer of its parent to null).

Example: delete 3 from this tree:

6

/ \

2 8

/ \

1 4

/

3

Case 2: x has only one child. Then remove it and adjust the appropriate

child link of x's parent to point to x's child.

Example: delete 4 from the tree above.

Case 3: x has two children. This is the complicated case. Find a value y

that is stored in a node with 0 or 1 children so that we can replace x with y

in x's node, and then remove the node that used to hold y. The value y

should be the SUCCESSOR of x (smallest value that is greater than x).

It is guaranteed to be the minimum element in x's right subtree (think about

why). Find it by going to x's right child, and then going left until hitting

a "dead end" (null pointer).

Example: delete 2 from the tree above.

Here is some code for removing from a BST:

void remove(int x) // public interface of the BST class

{

remove(x,root); // calls a private function, starting at root

}

Here's the recursive implementation of the private function.

Note that the second argument is a pointer that is passed by reference!

void remove(int x, BinaryNode\* & t) // private function

{

if (t == nullptr) // item not found, do nothing

return;

if (x < t->element) // continue search for x to the left

remove(x,t->left);

else if (x > t->element) // continue search for x to the right

remove(x,t->right);

else { // found x

if (t->left != nullptr && t->right != nullptr) { // x has 2 children

t->element = findMin(t->right)->element; // overwrite x with its successor

remove(t->element,t->right); // remove (duplicate) successor node

}

else { // x has 0 or 1 child

binaryNode \*oldNode = t; // remember node to be deleted

if (t->left != nullptr) // rearrange child pointers

t = t->left

else

t = t->right;

delete oldNode;

}

}

}

Worst-case running time is Theta(n), which occurs if the tree is a chain

and x is at the end of the chain.

A Few Words about Average Case Performance of Binary Search Trees:

-----------------------------------------------------------------

It can be proved that

(i) if all sequences in which n data elements are inserted are equally likely

and

(ii) no deletions are done,

then the average time per operation on an n-node tree is O(log n).

However, as the textbook says (p. 144): "deciding what "average" means

is generaly extremely difficult and can require assumptions that may or may

not be valid."

We'd like to have a search tree structure that ensures logarithmic time for

the operations in the WORST CASE. If the tree is "balanced", so that every

leaf is at depth O(log n), we can do so. But we need more complicated

algorithms for inserting and removing to maintain the balanced property.

Examples of such trees are shown next.

Section 4.4: AVL Trees

-----------------------

Definition: An AVL tree is a binary search tree in which for every node p,

the height of p's left subtree and the height of p's right subtree differ

by at most 1. (For convenience, define the height of the empty tree as -1.)

Example:

5

/ \

2 8

/ \ /

1 4 7

/

3

If we remove 7, is the result still an AVL tree?

So AVL trees are not perfectly balanced but they are close enough to ensure

that the height is O(log n), where n is the number of nodes.

It's actually not too hard to prove this: (argument adapted from Goodrich,

Tamassia and Mount, "Data Structures & Algorithms in C++" textbook)

Proposition: The number n of nodes in an AVL tree T with

height h is more than 2^{h/2}.

Assuming the proposition is true, we can rearrange and get

h < 2\*log n, i.e., h is O(log n).

Why is the proposition true? Let n(h) be the minimum number

of nodes in an AVL tree with height h.

- n(0) = 1 since an AVL tree with height 0 must have at least one

node (the root).

- n(1) = 2 since an AVL tree with height 1 must have at least two

nodes (the root and one child).

- n(2) = 4 since any binary tree with height 2 and only 3 nodes must

be a chain, which violates the AVL definition.

- For every h > 1, n(h) = 1 + n(h-1) + n(h-2) since in an AVL tree with

height h, at least one child of the root (which has height h) must

have height h-1 and the other child must have height h-2 or h-1.

n(h) > 2\*n(h-2) since n(h-1) >= n(h-2)

> 4\*n(h-4)

> 8\*n(h-6)

...

> 2^i \* n(h-2i)

...

Keep going until h-2i equals 0 or 1 (one of the base cases).

If h is even, then we'll reach 0 when i = h/2.

n(h) > 2^{h/2} \* n(0)

= 2^{h/2} since n(0) = 1

If h is odd, then we'll reach 1 when i = (h-1)/2.

n(h) > 2^{(h-1)/2} \* n(1)

= 2^{(h+1)/2} since n(1) = 2

In both cases (h even and h odd), n(h) > 2^{h/2}.

QED

In fact, it is possible to show that the height is at most 1.44\*log n.

(Convince yourself that the smallest possible height for a binary tree

with n nodes is floor(log\_2 n).)

Since the height of an AVL tree is O(log n), we can do the following

operations in O(log n) time:

\* contains(x)

\* findMin()

\* findMax()

How do we do insert in O(log n) time?

0. Store at each node the height of the node

1. First, do the basic BST insertion algorithm.

Since newly inserted nodes are leaves, they have height 0.

2. Start at the newly inserted node and follow parent pointers,

updating height information until reaching a node whose height

is unchanged or a node that is now unbalanced (its two children

have heights that differ by more than 1).

3. If a node is reached whose height is unchanged then no rearrangement

is needed.

4. If a node is reached that is unbalanced then rearrange using

ROTATIONS.

Suppose k2 is the lowest unbalanced ancestor of the new node.

....................................................................

Case 1: The new node is in the left subtree of k2.

Before the insertion:

---------------------

k2 k2 has height h (may have a parent)

/ \

k1 has height h-1 k1 [Z] subtree [Z] has height h-2

/ \

[X] [Y] subtrees [X] and [Y] have height h-2

since k1 remains balanced

(to be continued)

How do we do insert into an AVL tree in O(log n) time?

0. Store at each node the height of the node

1. First, do the basic BST insertion algorithm.

Since newly inserted nodes are leaves, they have height 0.

2. Start at the newly inserted node and follow parent pointers,

updating height information until reaching a node whose height

is unchanged or a node that is now unbalanced (its two children

have heights that differ by more than 1).

3. If a node is reached whose height is unchanged then no rearrangement

is needed.

4. If a node is reached that is unbalanced then rearrange using

ROTATIONS.

Suppose k2 is the lowest unbalanced ancestor of the new node.

....................................................................

Case 1: The new node is in the left subtree of k2.

Before the insertion:

---------------------

k2 k2 has height h (may have a parent)

/ \

k1 has height h-1 k1 [Z] subtree [Z] has height h-2

/ \

[X] [Y] subtrees [X] and [Y] have height h-2

since k1 remains balanced

Case 1.1: The new node is in the left subtree of k1, [X] becomes [X'].

After the insertion:

---------------------

k2 k2 has height h+1, UNBALANCED

/ \

k1 has height h k1 [Z] [Z] has height h-2

/ \

[X'] has height h-1 [X'] [Y] [Y] has height h-2

We can rebalance the tree using a SINGLE ROTATION:

- push k1 up to take k2's place

- make k2 the right child of k1

- move [Y] to become the left child of k2

After the rotation:

------------------

k1 k1 has height h, BALANCED

/ \

[X'] has height h-1 [X'] k2 k2 has height h-1

/ \

[Y] [Z] [Y] and [Z] have height h-2

Why is this correct?

\* We have restored the AVL tree balance property.

\* We have maintained the search tree ordering property: the

inorder traversal of the tree before the rotation is

[X'] k1 [Y] k2 [Z]

and the same is true after the rotation.

.......................................................................

Case 1.2: The new node is in the right subtree of k1, [Y] becomes [Y'].

After the insertion:

---------------------

k2 k2 has height h+1, UNBALANCED

/ \

k1 has height h k1 [Z] [Z] has height h-2

/ \

[X] has height h-2 [X] [Y'] [Y'] has height h-1

If we do the single rotation, we get:

k1 k1 has height h+1, still unbalanced

/ \

[X] has height h-2 [X] k2 k2 has height h

/ \

[Y'] has height h-1 [Y'] [Z] [Z] has height h-2

Instead, we need to do a double rotation. To describe it, we need

to look in more detail at [Y']. Suppose its root is k3, its left subtree

is [U] and its right subtree is [V]. (Different notation than in textbook

to be consistent with Case 1.1.)

k2 k2 has height h+1, UNBALANCED

/ \

k1 has height h k1 [Z] [Z] has height h-2

/ \

[X] has height h-2 [X] k3 k3 has height h-1

/ \

[U] [V] [U] or [V] has height h-2,

other one has height h-2 or h-3

- make k3 becomee the root of this subtree (take k2's place)

- push k2 down to become k3's right child

- k1 is left child of k3

- rearrange subtrees so that [U] is right child of k1

and [V] is left child of k2

k3 h1 has height h, BALANCED

/ \

k1 has height h-1 k1 k2 k2 has height h-1

/ \ / \

[X] has height h-2 [X] [U] [V] [Z] [V] has height h-2 or h-3

[U] has height h-2 [Z] has height h-2

or h-3

Why is this correct?

\* We have restored the AVL tree balance property.

\* We have maintained the search tree ordering property: the

inorder traversal of the tree before the rotation is

[X] k1 [U] k3 [V] k2 [Z]

and the same is true after the rotation.

.......................................................................

Case 2: The new node is in the RIGHT subtree of k2. This case and its

two subcases are the mirror image of Case 1: reverse the roles of "left"

and "right".

AVL Tree insert example:

Insert 3, 2, 1, 4, 5, 6, 7 in order. \* is the unbalanced node.

3 => 3 => 3\* => SR 2 => 2 => 2 => SR 2

/ / / \ / \ / \ / \

2 2 1 3 1 3 1 3\* 1 4

/ \ \ / \

1 4 4 3 5

\

5

=> 2\* => SR 4 => 4 => SR 4

/ \ / \ / \ / \

1 4 2 5 2 5\* 2 6

/ \ / \ \ / \ \ / \ / \

3 5 1 3 6 1 3 6 1 3 5 7

\ \

6 7

Now insert 16, 15, 14, 13, 12, 11, 10, 8, 9

=> 4 => 4 => DR 4 => 4

/ \ / \ / \ / \

2 6 2 6 2 6 2 6\*

/ \ / \ / \ / \ / \ / \ / \ / \

1 3 5 7 1 3 5 7\* 1 3 5 15 1 3 5 15

\ / \ / \

16 7 16 7 16

/ \

15 14

=> DR 4 => 4\* => SR 7

/ \ / \ / \

2 7 2 7 4 15

/ \ / \ / \ / \ / \ / \

1 3 6 15 1 3 6 15 2 6 14 16

/ / \ / / \ / \ / /

5 14 16 5 14 16 1 3 5 13

/

13

=> 7 => SR 7 => (insert 11 with a SR,

/ \ / \ insert 10 with a SR,

4 15 4 15 insert 8 with no rotation)

/ \ / \ / \ / \

2 6 14\* 16 2 6 13 16

/ \ / / / \ / /\

1 3 5 13 1 3 5 12 14

/

12

=> 7 => 7

/ \ / \

4 13 4 13

/ \ / \ / \ / \

2 6 11 15 2 6 11 15

/ \ / / \ / \ / \ / / \ / \

1 3 5 10 12 14 16 1 3 5 10\* 12 14 16

/ /

8 8

\

9

=> DR 7

/ \

4 13

/ \ / \

2 6 11 15

/ \ / / \ / \

1 3 5 9 12 14 16

/ \

8 10

....................................................................

Running time of insert algorithm is O(log n): Takes O(log n) time

to find the place to insert the new element, then go back up toward

the root checking for unbalanced nodes which takes O(log n) time.

If an unbalanced node is found, then do a single or double rotation

using O(1) time. Note that only one (single or double) rotation

is done; after that, the tree is balanced, since the subtree of

interest has the same height as it did before the rotation.

=================================================================

Removing a node from an AVL tree:

\* Do the basic BST removal.

\* Focus on the (structural) node that was removed (either the node

that contained the data item removed or the node that, before the

removal, contained the successor of the data item removed).

The removal of that node might unbalance the tree.

\* We have a similar set of 4 cases as for insert with one KEY DIFFERENCE:

After doing the rotation, the subtree of interest

might have height one less than it had before the removal.

\*\*\* so we have to proceed up the tree continuing to check for

unbalanced nodes.

The cases for remove:

- Let w be the parent of the (structural) node that was removed.

- Claim: At most one node in the tree becomes unbalanced by this

removal, and this node is on the path from w to the root.

- Go up the tree from w toward the root until finding an unbalanced

node, call it k2.

- Define k1 to be the child of k2 with larger height (it will NOT be

an ancestor of w).

Case 1: k1 is the left child of k2.

Case 1.1: The left child of k1 has larger height than the right child of k1

or their heights are equal.

Do a single rotation.

Case 1.2: The right child of k1 has larger height than the left child of k1.

Do a double rotation.

Case 2: mirror image of Case 1 (k1 is the right child of k2, etc.).

Example: Suppose a node in subtree [X] is removed, which reduces

the height of [X] from h-2 to h-3.

k2 k2 has height h, UNBALANCED

/ \

[X] has height h-3 [X] k1 k1 has height h-1

/ \

[Y] has height h-3 [Y] [Z] [Z] has height h-2

After the single rotation (Case 2.1), all the nodes in the subtree are

balanced, but the height has reduced by 1, which might cause an imbalance

further up the tree:

k1 k1 has height h-1, BALANCED

/ \

k2 has height h-2 k2 [Z] [Z] has height h-2

/ \

[X] has height h-3 [X] [Y] [Y] has height h-3

Note: If [Y] has height h-2, then after the single rotation, k1 has

height h (as did k2 originally) and there is no need to continue up the tree.

Running time of remove algorithm is O(log n): Takes O(log n) time

to do the basic BST deletion, then go back up toward the root checking

for unbalanced nodes. The path to the root has length O(log n). Even

if we have to do a rotation for each node on the path, the total time

is still O(log n).

Section 12.2: Red-Black Trees

-----------------------------

A popular alternative to the AVL tree is the red-black tree.

Definition: A red-black tree is a binary search tree in which

every "missing" child is replaced with a dummy sentinel node which holds

no data (all the leaves are sentinels, every data node has exactly two

children, every sentinel node has 0 children), and

1. every node is colored red or black such that

2. the root is black

3. the children of a red node are black

4. every leaf (sentinel) node is black

5. every path from the root down to a leaf has the same number of black nodes.

Example: (\* means red, otherwise black)

30

/ \

15 70

/ \ / \

10 20 60\* 85

/\ / \ / \ / \

5\* . . . 50 65 80\* 90\* . indicates a sentinel node

/ \ / \ /\ /\ /\

. . 40\* 55\* . .. . . .

/\ /\

. . . .

Check that every leaf has 4 black ancestors (including itself as an

ancestor) and that no red node has a red parent.

Claim: The height h of a red-black tree T with n nodes is at most 2\*log(n+1).

Proof:

Let h' be the number of black nodes on each path from the root of T to a leaf.

T has the minimum number of nodes for a fixed value of h' if it has no red

nodes. In this case T is a complete binary tree with height h' and thus

has 2^{h'} - 1 nodes.

So n >= 2^{h'} - 1, which implies h' <= log(n+1).

Since the children of a red node are black, h <= 2\*h'.

Thus h <= 2\*log(n+1).

QED

So the depth isn't quite as small as for an AVL tree.

However, the red-black tree insert and remove algorithms only require

O(1) STRUCTURAL changes to the tree, instead of Theta(log n) for an

AVL tree.

Searching:

---------

The searching algorithm is the same as for the basic binary search tree

but takes O(log n) time in the worst case.

Inserting:

---------

First, do the basic binary search tree insertion.

Color the newly inserted node X red.

If X's parent P is black, then we are done.

Suppose P is red. Then P cannot be the root (since root must be black).

Let G be P's parent. G must be black (and might have a parent)

Let S be P's sibling (might be a sentinel).

Case 1: S is black.

Case 1.1: X and P are both left children or both right children.

I'll just draw the picture for the left case.

G

/ \

P\* S

/ \ [C] [C] is the subtree rooted at S

X\* [B] [B] is the subtree rooted at P's

[A] (unnamed) right child

[A] is the subtree rooted at X

Do a single rotation and recoloring to get:

P

/ \

X\* G\*

[A] / \

[B] S

[C]

Check correctness:

- rotation does not change the inorder traversal

- each leaf in [A], [B] and [C] has the same black depth after the rotation

as it did before the rotation.

- no red node has a red parent.

Case 1.2: X is a right child and P is a left child (or vice versa).

G => X

/ \ / \

P\* S P\* G\*

/ \ [C] / \ / \

[A] X\* [A] [B1] [B2] S

/ \ [C]

[B1] [B2]

Check correctness:

- rotation does not change the inorder traversal

- each leaf in [A], [B1], [B2], and [C] has the same black depth after

the rotation as it did before the rotation.

- no red node has a red parent.

Case 2: S is red.

G

/ \

P\* S\*

/ \ [C]

X\* [B]

[A]

We do this recoloring:

G\*

/ \

P S

/ \ [C]

X\* [B]

[A]

If G is the root, we change its color back to black, which increments the

black depth of all leaves by 1, and we are done.

Otherwise (G is not the root), the recoloring does not change the

black depth of the leaves in [A], [B] and [C].

If G's parent is black, we are done.

If G's parent is red, we move up the tree (think of G as X and G's

parent as P) and see if we are in Case 1 or Case 2. As long as we are

in Case 2, we keep recoloring and moving up. If we ever hit Case 1,

we do a restructuring and recoloring and we are done. So we only have

to do one restructuring, although we might have to do O(log n)

recolorings.

The way the insert algorithm is described so far, we make one pass

down the tree to find the insertion point and then another pass back

up the tree to fix the structure and colors. An optimization is just

to make a single pass down the tree (thus avoiding the need for parent

pointers or keeping the addresses to the parents on a stack), and to

proactively change the colors and structure. (Cf. Section 12.2.2.)

Example: The example tree from above is the result of inserting, in order,

10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55. (Not drawing the

black sentinel leaves.)

Next insert 45:

30

/ \

15 70

/ \ / \

10 20 60\* 85

/ / \ / \

5\* 50 65 80\* 90\*

/ \

40\* 55\*

45 becomes the right child of 40 and is colored red.

Since 40 and 55 are both red, we are in Case 2, so we recolor 40

and 55 as black, and make their parent 50 red:

30

/ \

15 70

/ \ / \

10 20 60\* 85

/ / \ / \

5\* 50\* 65 80\* 90\*

/ \

40 55

\

45\*

Since 50 and its parent 60 are both red, we are not done.

Since 50's sibling 65 is black, we are in Case 1.

Since 50 and 60 are both left children, we are in Case 1.1.

After doing the rotation and recoloring, we get:

30

/ \

15 60

/ \ / \

10 20 50\* 70\*

/ / \ / \

5\* 40 55 65 85

\ / \

45\* 80\* 90\*

Removal: (optional)

-------

(Presentation is from Goodrich, Tamassia, and Mount textbook.)

Assume that we have dummy sentinel nodes for all missing children and

they are black. So all the leaves are sentinels.

First do the basic binary search tree removal operation, so we can

consider removing a node v with at least one leaf child w, i.e., v has

at most one "real", non-sentinel, child.

Let r be the sibling of w and x be the parent of w. We remove v and w

and r replaces v as a child of x. (Picture shows r becoming a right

child; there is a mirror image case for when r becomes a left child.)

... ...

x x

/ \ => / \

v r

/ \

w r w is a sentinel, r might be a sentinel

Now let's worry about the colors.

First, the easy cases.

\* If v was red, then r is black and all is fine.

\* If r is red, then v was black, so recolor r to be black and all is fine.

Now the harder case: both v and r were black, so when we remove v,

we've messed up the black depth property. We have to make more substantial

changes. Think of r has being "double black". There are 3 cases.

Case 1: r's (new) sibling y is black and has a red child z.

(Picture shows z being a left child; there is a mirror image situation

when z is a right child.)

...

x x can be either red or black

/ \

y B r BB BB indicates r is "double black"

/ \ / \

z R [T2] [T3] [T4]

/ \

[T0] [T1]

After restructuring and recoloring:

...

y y's color is x's former color

/ \

z B x B

/ \ / \

[T0] [T1] [T2] r B

/ \

[T3] [T4]

Now everything is fixed.

Case 2: r's (new) sibling y is black and both y's children are black.

...

x x can be either red or black

/ \

y B r BB BB indicates r is "double black"

/ \ / \

B B

Recolor by trying to move r's extra black upward:

... if x used to be R, now it is B

x if x used to be B, now it is BB

/ \

y R r B now r is just B and y is R

/ \ / \

B B

If x is now B, then we are done.

If x is now BB and it's the root, then make x just B, which decreases

the black depth of each leaf by 1.

If x is now BB and it's not the root, then we have to continue moving

up the tree to try to get rid of the BB, by considering the 3 cases.

Case 3: r's (new) sibling y is red.

...

x B x has to be B since it has a R child

/ \

y R r BB

/ \ / \

z B [T2] . . note that root of T2 must be black

/ \

[T0] [T1]

Do an "adjustment", which restructures and recolors to get this:

...

y B y is now B

/ \

z B x R x is now R

/ \ / \

[T0] [T1] [T2] r BB recall root of T2 is black

/ \

. .

We didn't get rid of the BB problem, but we've shifted into Case 1 or 2

since r's sibling is now black. If we're in Case 1, then after one

restructuring and recoloring we are done. If we're in Case 2, the BB

problem will not reappear since r's parent is red and so after

one recoloring we are done.

The total time for removal is O(log n), since at each level in the tree

we do O(1) work and there are O(log n) levels. Furthermore, at most two

restructurings are done.

Example:

14

/ \

7\* 16\* \* means red

/ \ / \

4 12 15 18

/ \ /\ /\ / \

3\* 5\* . . . . 17\* . sentinel leaf indicated by .

/\ / \ /\

. . . . . .

remove 3 => easy:

14

/ \

7\* 16\*

/ \ / \

4 12 15 18

/ \ /\ /\ / \

. 5\* . . . . 17\* .

/ \ /\

. . . .

remove 12 => causes double black sentinel leaf (marked as .BB) and restructuring:

14 => 14

/ \ / \

7\* 16\* 5\* 16\*

/ \ / \ / \ / \

4 .BB 15 18 4 7 15 18

/ \ /\ / \ /\ /\ /\ / \

. 5\* . . 17\* . . . . . . . 17\* .

/ \ /\ / \

. . . . . .

remove 17 => easy:

14

/ \

5\* 16\*

/ \ / \

4 7 15 18

/\ /\ /\ /\

. . . . . . . .

remove 18 => causes double black sentinel leaf and recoloring of 16 and 15:

14 => 14

/ \ / \

5\* 16\* 5\* 16

/ \ / \ / \ / \

4 7 15 .BB 4 7 15\* .

/\ /\ /\ /\ /\ /\

. . . . . . . . . . . .

remove 15 => easy:

14

/ \

5\* 16

/ \ / \

4 7 . .

/\ /\

. . . .

remove 16 => causes double black sentinel leaf, restructuring, and then

recoloring:

14 => 5 => 5

/ \ / \ / \

5\* .BB 4 14\* 4 14

/ \ /\ / \ /\ / \

4 7 . . 7 .BB . . 7\* .

/\ /\ /\ /\

. . . . . . . .

Section 4.6: Tree Traversals (Revisited)

-----------------------------------------

Applications of different kinds of traversals on any kind of search

tree (basic, AVL, red-black):

\* inorder traversal: outputs the elements of the tree in sorted order.

void printTree() const { // public function

if (isEmpty()) cout << "empty tree";

else printTree(root); // call private printTree function

}

void printTree(BinaryNode\* t) const { // private function

if (t != nullptr) {

printTree(t->left);

cout << t->element << endl;

printTree(t->right);

}

}

\* postorder traversal: compute the height of each node in the tree;

Before calculating the height of a node, we must know the heights of

its children.

int height(BinaryNode\* t) {

if (t == nullptr) return -1;

return 1 + max(height(t->left),height(t->right));

}

\* preorder traversal: compute the depth of each node in the tree.

(You can write the code.)

Remember that all these traversals take Theta(n) time for an n-node

tree, since the work done when visiting a node is constant time.

======================================================================

\*\* skip Section 4.7 on B-Trees \*\*

Key new idea here is to store many more than one data item in each

tree node, which reduces the height (by a constant factor).

======================================================================

Section 4.8: Sets and Maps in the C++ STL

------------------------------------------

This chapter and the next present implementations that can be used for

several similar abstract data types. The similarity between the ADTs

is that they are all collection of elements. Elements can be put

into the collection, removed from the collection, and searched for in

the collection.

There are three main ways in which these ADTs can vary:

(1) Is the element type "monolithic" (e.g., int) or is it better viewed as

a pair consisting of a "key" and a "value" (e.g., a Student class where

the key is the SSN and the value is all the other info about the student)?

In the first case, we can refer to each element as a key (which has no

associated value).

(2) Must the keys be unique or can different elements have the same key?

(3) Are the elements kept in order w.r.t. some relation on the keys or not?

Warning! Different programming languages, books, papers, people, etc.

use inconsistent terms for these concepts. So always double-check.

First, assume elements are (key,value) pairs.

unique keys duplicate keys allows

------------------------------------------------

unordered |

| unordered\_map (C++) unordered\_multimap (C++)

--------------------------------------------------

ordered |

| map (C++) multimap (C++)

|--------------------------------------------------

Second, assume elements are just keys (with no associated values).

unique keys duplicate keys allows

------------------------------------------------

unordered |

| unordered\_set (C++) unordered\_multiset (C++)

--------------------------------------------------

ordered |

| set (C++) multiset (C++)

|--------------------------------------------------

Usual C++ implementations are

- hash tables for unordered\_set and unordered\_multiset (Ch 5)

- search trees for (ordered) set and (ordered) multiset (Ch 4)

In fact, the gcc C++ compiler implements the STL map with a red-black tree:

https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl\_map.h

======================================================================

Summary:

--------

\* Search trees are an extremely important data structure and are used

in many applications, especially those that deal with a collection of

data items and need to insert into, remove from, and search the collection.

\* Recursive algorithms for the search tree operations are easier to

understand and debug, while iterative (non-recursive) ones are somewhat

faster (not asymptotically, but with respect to constant factors).

\* Balanced search trees (e.g., AVL, red-black) provide worst-case

logarithmic time for the main operations, while the basic binary search

tree has simpler insert and remove algorithms.

\* The "search-tree sort" algorithm schema works by inserting the items

to be sorted into a search tree and then outputting the items in the tree

using inorder traversal.