

1999 APS105S Lectures 37/38: Binary Trees

Last Day — queues!

Today — doubly-linked lists, binary trees

Projects

- GREAT JOB! I was really impressed with the effort and results!

Quick Review: Stacks and Queues

Stacks

- LIFO
- push(), pop(), peek(), isEmpty()
- can use linked list OR arrays to implement
 - showed you a version using arrays
 - limitation of array: fixed size

Queues

- FIFO
- enqueue(), dequeue(), peek()?, isEmpty()
- can use linked list OR arrays to implement
 - showed you using a simple “extension” to linked lists
 - limitation of array: fixed size

Comparison

- similar methods, different policies
- can use arrays or linked lists
- similar limitations
- both are ADTs

Improving Linked Lists: Binary Search?

Finding an Element

- recall the telephone book
- it's like an “array” of names with phone #
- we can do a binary search to find a name, lookup phone #
- suppose we wish to insert a new name in the book
 - all other names must move down one
 - very time-consuming operation, even on a computer
- use a linked list instead

can we still do a binary search?

- we must split the list in half
- step to find the middle element is still $O(n)$

what about a doubly-linked list?

- still $O(n)$

we cannot binary search on a linked list!

give up, try something different

Improving Linked Lists: Doubly-Linked

Deleting an Element

Given

- a reference to an object in a linked list to delete

Problem

- to delete, must change the pointer of the object *before* it

Solution 1

- go to head. is this the object?
- no, go to next. is this the object? ...
- stop one link before finding the object-to-be-deleted

Time Complexity?

- doing a search in the linked list, $O(n)$

Solution 2

- we always know the “next” object
- why not know the “previous” object too?
- called a doubly-linked list

Time Complexity?

- no more searching, just do it!, $O(1)$

Advantages?

- fast delete
- important for LARGE lists
- can now walk backwards through the list!
 - start with a ‘tail’, to ‘prev’ until you find ‘null’ (at head)
 - going backwards is sometimes very useful

Disadvantages?

- difficult to program (you thought linked lists were hard!)
- uses extra memory for all the ‘prev’ references
- worth getting it right?

Binary Trees

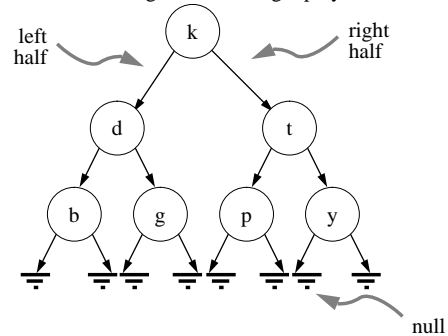
can we organize things better?

- have a pointer to the “middle” instead of “next” and “prev”?

- reorganize phone book
 - no longer has a first page and a last page
- now it only has a “middle page”
- each middle page must refer to the “left half” and “right half”
- the “left half” is really the “middle of the left half”

Example

- given names that begin with: b d g k p t y



- -b- =d- >g- <k- <-p- =t- >y-
- ^-----/\-----^

- called a tree, specifically binary tree (why? hint: binary = 2)

- first node is called the root
- it contains
 - 1/ some data value (eg, k)
 - 2/ reference to a “left half” ... (eg, new object containing d)
 - 3/ reference to a “right half” ... (eg, new object containing t)

```
class BinTreeElem
{
    String      name;
    BinTreeElem left;
    BinTreeElem right;
}
```

- the “left half” is really another “subtree” where
 - the value is the middle of the subtree
 - it contains a reference to a “left half”
 - and a “right half” a naturally recursive structure ;)
- what about when we reach the “ends” of the tree?
- no more left/right halves... call these “leaves”
- use “null” to say “no more subtrees” or branches
- “middle nodes” — sometimes called branches

Kind of Binary Tree: “binary search tree”

“binary search tree” is a very clever way to organize your data we can use it to do a binary search:

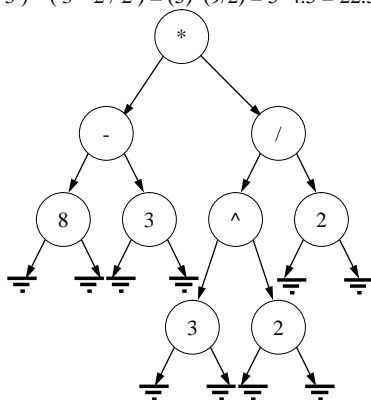
```
binSearch( node, searchKey )
if( node == null ) return null;
else if( node.value < searchKey )
    return binSearch( node.right, searchKey )
else
    return binSearch( node.left, searchKey )
```

Another Binary Tree: “expression tree”

- given an equation, eg: $1 + (2 - 3) * 4 ^ 5$
- how do you read this into a program?
- how do you store it? evaluate it?

Example

• $(8 - 3) * (3 ^ 2 / 2) = (5) * (9/2) = 5 * 4.5 = 22.5$



Notice

- not necessarily minimum, full, or balanced!
- the “leaf nodes” are always numbers
- the “branch nodes” are always operators
- each subtree evaluates to a value before it is included in parent

binary search trees: recursive! always cutting our work in half!

- doubly-linked list contains same amount of info for each object (left obj, right obj), but we can’t binary search it!
- binary search trees **must have** the following **order properties**:
 - left child \leq ‘self’ $<$ right child
 - all left children \leq ‘self’ $<$ all right children
- as well, they must be organized as a tree for efficient searching
 - eg: a tree has degenerated to a list if all its “left” references are null
- ideally, binary search trees should be:
 - **minimum depth** has minimum number of levels
 - **balanced** similar size on left and right halves
- if these aren’t true, the tree will be less efficient
 - extra levels means “one extra search step”
 - unbalanced means we aren’t cutting our work in half (remember when quicksort was not balanced? $O(N^2)$)
- if the binary search tree is “ideal”, searches are $O(\log N)$
 - like array binary search, each step halves the candidates
 - else, searches may be $O(N)$ (in worst case)
- the example above, the binary tree is also **full**
 - adding one extra element to the tree forces us to use one more level
 - for n levels, there are $2^n - 1$ elements in a full tree
- if we insert or delete a new item in the tree
 - if binary search tree, must preserve **ordering property**
 - try to preserve the “balance” and “minimum depth”
 - there are fancy insert, delete tricks to do this
 - think about how you might do this...

Building a Calculator

Construct a program to read an expression like this:

$(8 - 3) * (3 ^ 2 / 2)$

and compute the answer.

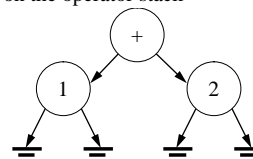
Reading the Equation

- want to read in the equation, build the expression tree
- read one “element” at a time: (then 8 then - then 3 ...
- respect order-of-operations
- as we scan the equation, we will place “seen items” on a stack
- use 2 stacks: one for “numbers” one for “operators”

Example 1

$1 + 2 - 3$

- see 1, push 1 on number stack
- see +, push + on operator stack
- see 2, push 2 on operator stack
- see -, same precedence as +, we can now legally do $1 + 2$
 - pop 2, 1 from number stack
 - pop + from operator stack
 - combine, or “bind” them to a small tree
 - push the “tree” as a single “number” on the number stack
- push the - on the operator stack

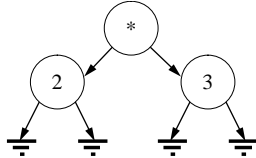


- see 3, push 3 on number stack
- no more input, “bind” - on opr stack to values in num stack

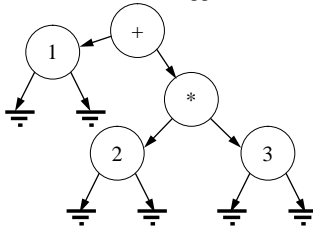
Example 2

1 + 2 * 3

- see 1, push 1 on number stack
- see +, push + on operator stack
- see 2, push 2 on operator stack
- see *, higher precedence than +, cannot add yet!
- see 3, push 3 on number stack
- no more input
 - “bind” operators on operator stack to values in num stack
 - pop 3 and 2 off number stack
 - pop * off operator stack
 - “bind” 2 * 3 to result in a small tree



- push the small tree 2 * 3 on number stack
- pop 2*3 and 1 off number stack
- pop + off operator stack
- bind 1 + 2*3 to result in a bigger tree



Algorithm and Program Notes

- need to know if one operator has higher priority than another
 - write your own “priority” method
- cannot handle leading “-” sign for unary minus
 - pretend the “-” is part of the number, e.g. “-1”
 - use spaces to separate all numbers and operators
- store numbers and operators as a string
- convert to actual values at “last minute” as needed
- must put elements in
 - linked list (for a stack)
 - a binary tree (for expression tree)
 - need: next, left, right references
 - stack uses next reference
 - binary expression tree uses left, right references

General Case: Shunting Yard Algorithm

1 + 2 * 3 - (4 - 5 ^ (6))

Scan the equation left-to-right, for each element:

- if a number, push it on number stack
- if an operator, compare priority with one on top of stack
 - **WHILE** the top-of-stack operator is same or higher priority
 - stack operator should be done first
 - “bind” top 2 values in number stack to operator on stack
 - push the tree back on number stack
 - go to top of while, repeat as necessary
 - now stack operator is lower priority than new operator
 - push new operator on the operator stack
- if no more input
 - operators and numbers remain on stacks
 - top of operator stack has highest priority, bottom operator has lowest priority
 - while operator stack is not empty
 - bind top operator to top 2 numbers
 - replace top 2 numbers with one resulting tree
- what about parentheses ?
- relatively easy to handle...
 - (has lowest priority, put on stack as a marker
 -) has highest priority
 - when seen, bind everything on stack up to matching (
 - this replaces entire (...) expression with a tree (treated as a single number on the number stack)
- easy way to handle “no more input” case
 - place (...) around entire equation
 - when last) is seen, everything up to opening (is bound, i.e. the whole equation

Traversing a Tree

- 1 way to “walk” or traverse a linked list (visit all elements)
 - follow next!
- 3 ways to “walk a tree”
 - in order, preorder, postorder!
 - in order
 - self is in middle
 - visit left, then self, then right
 - see toInfixString() method in code
 - this is the approach taken to print an expression tree as a “normal” equation (must add parentheses to enforce proper ordering)
 - preorder
 - self is first
 - visit self, then left, then right
 - see toPrefixString() method in code
 - this is the order taken during a binary search (though you usually take only the left or the right branch, not both)
 - postorder
 - self is last
 - visit left, then right, then self
 - see toPostfixString() method in code
 - this is the approach taken to evaluate an expression tree (you must evaluate the left and right sides before combining them)
 - left and right visits are actually recursive calls

See Calc.java source code!

```

// APS105S
// Calculator Example Program.
//
// This program uses stacks and binary trees to read a
// "normal-looking" equation from the user, store it internally,
// and compute an answer (in double format).
//
// The elements of an equation must be comprised as follows:
// - numbers will be treated as double values
// - double operators: * / + - and ^ (exponent/power)
// - parentheses: ( and )
//
// Additional rules:
// - no variables like "x"
// - there must be a space between every element, examples:
// 1 + 2
// 1 + ( 2 - 3 ) * 4
// 1 + 2 * 3.001 ^ 4.2
// 1 + 2 * 3.001 ^ 4.2
//
// - the equation must fit on one line
// - parentheses must be properly matched; don't forget any!
// - order of operations is respected -- use ( ) if you need to
// - same-priority operators are evaluated left-to-right
//
// Enter an empty equation to quit.
//
// This program does not check for 100% valid input. Some input
// appears valid to it, but is incorrect. It does not gracefully
// handle errors when parentheses are mismatched.
//
// Errors that are unrecognized/ignored:
// 1 2 + 3      <- doesn't notice error, ignores the '1'
// 1 + 2 * 3 )  <- doesn't notice error
// 1 + ( 2 * 3  <- prints error, tries its best.
//
// The expression is read in using the Shunting Yard Algorithm.
// For details, please see the accompanying lecture notes.
class Elem
{
    String val; // value: contains either an integer (leaf nodes)
               // or an operator (branch nodes)
    Elem next; // for the stack: linked-list next pointers
    Elem left; // for binary tree: left and right pointers
    Elem right; //
    public Elem( String s )
    { val = s; }

    // methods to "walk the tree" for us
    // there are 3 ways to walk the tree: pre-order, in-order, and post-order
    // these 3 methods generate:
    // - a prefix equation,
    // - an infix equation (what you're normally used to), and
    // - a postfix equation (reverse polish notation)
    // we'll use commas to separate numbers in the prefix/postfix form.
    // we'll add parentheses in the infix form to preserve order of operations

    public String toPrefixString()
    {
        String str = val;
        if( left != null )
            str = str + "," + left.toPrefixString();
        if( right != null )
            str = str + "," + right.toPrefixString();
        return str;
    }
}

public String toInfixString()
{
    String str = "";
    if( left != null )
        str = "(" + left.toInfixString();
    str = str + val;
    if( right != null )
        str = str + right.toInfixString() + ";";
    return str;
}

public String toPostfixString()
{
    String str = "";
    if( left != null )
        str = left.toPostfixString() + ",";
    if( right != null )
        str = str + right.toPostfixString() + ",";
    return str;
}

// val is an operator. compute (x val y),
// and return the answer.
public double operate( double x, double y )
{
    if( val.equals("^") )
        return Math.pow( x, y );
    else if( val.equals("*") )
        return x * y;
    else if( val.equals("/") )
        return x / y;
    else if( val.equals("++") )
        return x + y;
    else if( val.equals("--") )
        return x - y;
    else {
        System.out.println("Error, unknown operator: " + val);
        return 0.0;
    }
}

// walk the tree, evaluating the equation.
// this is one in post-order, because the left and right
// side values must be known to operate on them.
public double eval()
{
    double answer;
    if( left == null && right == null ) {
        // no children, must be a number
        // convert 'val' to a double
        try {
            answer = Double.valueOf(val).doubleValue();
        } catch( NumberFormatException e ) {
            answer = 0.0;
        }
        return answer;
    } else {
        // has children, must be an operator
        double leftValue = left.eval();
        double rightValue = right.eval();
        answer = operate( leftValue, rightValue );
        return answer;
    }
}

// End of "class Elem"

```

```

// You've all seen stack code before right?
// This implementation is based on linked lists.
// Well keep track of the top of the stack with a
// topPtr reference.

class Stack
{
    Elem topPtr;

    public Stack()
    {
        topPtr = null;
    }

    public void push( Elem e )
    {
        e.next = topPtr;
        topPtr = e;
    }

    public Elem pop()
    {
        Elem e = topPtr;
        if( topPtr != null )
            topPtr = topPtr.next;
        return e;
    }

    public Elem peek()
    {
        return topPtr;
    }

    public boolean isEmpty() {
        return topPtr == null;
    }
} // End of "class Stack"

public class Calc
{
    public static Stack oprStack;
    public static Stack numStack;

    // give the priority of the operator. higher number = higher priority,
    // which is more tightly binding.
    public static int priority( String opr ) {
        if( opr.equals("^") )
            return 3;
        else if( opr.equals("*") || opr.equals("/") )
            return 2;
        else if( opr.equals("++") || opr.equals("--") )
            return 1;
        else // don't know what it is, lowest binding priority
            return 0;
    }

    // break up the equation into two pieces: the first element (word)
    // and the "rest" of the equation after the first word is removed.
    // we'll assume spaces separate *all* of the terms and operators in
    // an equation
    public static Elem getFirstElem( String equation )
    {
        int i = equation.indexOf(" ");
        String firstPart;
        if( i > 0 )
            firstPart = equation.substring(0,i+1).trim();
        else
            firstPart = equation;
        return new Elem( firstPart );
    }

    public static String removeFirstWord( String equation )
    {
        int i = equation.indexOf(" ");
        if( i > 0 )
            return equation.substring(i+1).trim();
        else
            return "";
    }

    // take the top operator off the opr stack
    // take its two arguments off the number stack
    // bind them together into a tree
    // save the tree back on the number stack
    public static void bindTopOpr()
    {
        if( !oprStack.isEmpty() ) {
            Elem subExpr = oprStack.pop();
            subExpr.right = numStack.pop();
            subExpr.left = numStack.pop();
            numStack.push( subExpr );
        }
    }

    // we previously stacked a "(" on oprStack, and
    // we just saw a ")" in the equation input.
    // bind everything from opening "(" to matching ")" in a tree,
    // and then put this tree back on the numStack
    // return early if there is an error.
    public static void doBrackets()
    {
        while( true ) {
            Elem opr = oprStack.peek();

```

```

if( opr == null ) {
    // oops, we ran out of equation terms and didn't find matching "("
    System.out.println( "Error, missing (" );
    return; // there was an error, quit early
} else if( opr.val.equals("(") ) {
    // remove "(" from oprStack, its no longer needed
    oprStack.pop();
    return; // there was no error, normal quit
} else {
    // collapse top 2 subtrees on numStack and top operator
    // places resulting tree back on numStack
    bindTopOpr();
}

// store the new operator 'opr' on the stack because we haven't seen its
// right-hand argument yet. before we do that however, we must check if
// the previous operator was higher priority; if so, it must bind to its
// two arguments now.
public static void pushNewOpr( Elem opr )
{
    Elem prevOpr = oprStack.peek();
    while( prevOpr != null
        && priority(prevOpr.val) >= priority(opr.val) )
    {
        // if prev opr was higher priority than this one,
        // collapse the previous expression into a tree
        bindTopOpr();
        prevOpr = oprStack.peek();
    }
    // now push our current operator on the oprStack
    oprStack.push( opr );
}

// parse equation, one term at a time. (read comment at the top of this
// file to see what the input looks like). the terms get pushed on stacks
// and "binding" is done when we can collapse a subexpression down into a
// tree.
public static void parseEquation( String equation )
{
    // start equation with an extra "("
    oprStack.push( new Elem("(") );

    while( ! equation.equals(" ") ) {
        Elem firstPart = getFirstElem( equation );
        equation = removeFirstWord( equation );

        if( firstPart.val.equals("(") ) {
            // pop everything until the previous ( on the stack
            doBrackets();
            // one of: ^ * / + -
            pushNewOpr( firstPart );
        } else if( firstPart.val.equals("(") ) {
            // don't evaluate prev opr yet, wait till ) is seen
            oprStack.push( firstPart );
        } else {
            /* have a number */
            numStack.push( firstPart );
        }
    }

    // end equation with an extra ")"
}

// this automatically does final cleanup for us :-))
doBrackets();
}

// print the equation, as stored in the tree, 3 different ways.
// evaluate the equation and print the answer.
public static void printResult()
{
    Elem eqn = numStack.pop();
    System.out.println( "Printing infix version:" );
    System.out.println( "\t" + eqn.toInfixString() );
    System.out.println( "Printing prefix version:" );
    System.out.println( "\t" + eqn.toPrefixString() );
    System.out.println( "Printing postfix version:" );
    System.out.println( "\t" + eqn.toPostfixString() );
    System.out.println( "Evaluating expression:" );
    System.out.println( "\t" + eqn.eval() );
    // sometimes errors leave things on the stacks,
    // so this forces the stacks to be emptied.
    while( ! numStack.isEmpty() )
        numStack.pop();
    while( ! oprStack.isEmpty() )
        oprStack.pop();
}

// repeatedly ask for a new equation, evaluate it, print answer.
// stop when the equation input is empty.
public static void main( String[] args )
{
    String input;
    String output;
    numStack = new Stack();
    oprStack = new Stack();

    while( true ) {
        // ask the user for an equation
        System.out.print( "\nEnter equation:\n\t" );
        input = Stdin.getString();

        input = input.trim(); // remove whitespace before and after
        if( input.equals("") )
            break;
        parseEquation( input );
        printResult();
    }
    System.out.println( "Done." );
}

// End of "Class Calc"
}

```