Merge Operation Programming

MergeSort Programming

Merge Sort Demo.

Selection Sort Demo.

Project 3 Data Structure.

Project 3 Management.

You are uncomfortable with recursion.

Recursion—tracing recursive programs. Do self-test and some DSO "projects".

Midterm next week to cover subjects of Labs and Projects 1-3.

CSI 310: Lecture 13 (Spr 05)
Trees, Recursion, Expressions, Stacks are closely related.

**Definition:** A function is **recursive** means the body of the function sometimes, when the function runs, calls the same function, either directly or indirectly.

```c
int fact(int n)
{ if(n==0) return 1;        else return fact(n-1)*n;  }
```

First, note **n** is an automatic, or LOCAL (Extant) VARIABLE.

Each time a function is CALLED, an ACTIVATION RECORD is created.

The EVENT of “Calling one function once” is an ACTIVATION.

The storage used for AUTOMATIC VARIABLES is in the activation record. **Different ACTIVATIONS—Different AUTOMATIC VARIABLES!!**
RETURNED TO.

4. Now, automatic variables are in the activation record of the activation

RETURNED TO.

3. The ACTIVATION whose CALL originally created this activation is

"goes away".

2. This ACTIVATION'S ACTIVATION RECORD

I. The return value (if any) is saved for use by the caller.

...when a function ACTIVATION executes the return statement...

University at Albany Computer Science Dept.
But, running $\text{fact}(0)$ calculates $0! = 0$ is true and returns 1.

Then computes $720! = 5040$ and returns it.

For example, running $\text{fact}(7)$ computes 7! = 5040, which returns

```c
{ if(n==0) return 1; else return fact(n-1)*n; }
int fact(int n)
```
6. Write interfaces and pre/post conditions for a split function, `mergesort()`.

5. Follow instructions to implement sort algorithm timing.


2. Input phase linked list builder and `printforward()` for testing/preamble.

1. Input phase line reader and skeleton `main()`.

Project 3 Management
test, debug, final testing, project completion/turnin.

10. Implementation merge sort timing, perform final review of project specifications.

9. Finish merge sort() to use split and merge; test and debug.

8. Design, code, test and debug the function

7. Begin merge sort() to provide test input to merge sort().

6. Dispatch to provide test input to merge sort().

5. Before coding the recursive calls, use input phase and command merge sort, before coding the recursive calls. Use input phase.

4. Hint: use your print split code for testing the "divide" step of merge sort().

3. Begin merge sort() body. Copy OR rewrite your split code to put in

2. Design, code, test and debug the function

1. Implementation merge sort timing, perform final review of project specifications.
```cpp
{ return count >> A.end; // A sorted string was printed.
  now, A[0..nch-1] is sorted
}
now, A[t] has the smallest char from A[t..nch-1]
{
  /* [?] A[t], A[] swap */
  if (A[t] < A[?])
    for (int j = t+1; j > nch; j++)
      for (int i = 0; i < nch-1; i++)
        if (strftime(A, asize, return 1; int nch = strlen(A));
      if (strftime(A, asize) return 100; char A[asize] =
        )
    main()
using namespace std;
#include <string>
#include <iostream>
#include <fstream>
Selection sort demo: processes chars within the array A.
```
We will then illustrate the (recursive) merge sort half.

We now illustrate what half your Project 3 work must do.

It uses the selection sort algorithm for sorting.

This program manipulates characters as if they were numbers.
Sample list of lines to sort:

- Cat
- Aardvark
- Bat
- Ape
- Caterpillar
- Dog
- Ant
- Zebra
How can we very efficiently swap the strings in the nodes pointed to by I and j?
Computer DOES NOT copy chars nor node pointers!

Swap the values in the 2 data fields of the nodes pointed to by I and J.

pT = I->data; I->data = J->data; J->data = pT;

j

I

TAIL

HEAD

pT

I

J

TAIL

HEAD

null

null

null

null

null

null

null

null
3. Combining the two sorted groups into one large sorted list. This combining

**INDEPENDENTLY!**

2. Sort each of these smaller groups (by recursive calls). That means

size, "Split" in project 3.

1. Divide the elements to be sorted into two groups of equal (or almost equal)

arrays, which is easier.

paradigm or pattern is applied to the problem of sorting a sequence: (read the

Mergesort is the algorithm that is invented when the **divide-and-conquer**
(1) // (or p == NULL, if you choose option (1))
// as defined by const

(2) // the original C-strings are returned in non-decr. lexico-grammatical order,

post: The return value points to a linked list holding

your choice: (1) or p == NULL, (2) p != NULL

pre: p points to a linked list of C-strings.

node * MERGESORT(node * p) {

node * node { node * node; char * data; }

Please declare and code these:

University at Albany Computer Science Dept.
{ return Solution;
{
Solution = MERGE (Ans1, Ans2);
node * Ans2 = MERGESORT (Subproblem2);
node * Ans1 = MERGESORT (Subproblem1);

RECURSE one or more times:
Subproblem2 = Instance;
//SPILL removed half the original list.
Subproblem1 = SPILL (Instance);
node * Subproblem1, Subproblem2;
}
else
{
Solution = Instance; //TRIVIAL SORT OF A 1-ELEMENT LIST

if (Length(Instance) == 1)
    node * Solution;
node * MERGESORT(node * Instance)
Divide and Conquer pattern APPLIED TO SORRING
of the half that WAS REMOVED from the linked list.

(2) The return value is the addr. of the first node the nodes removed.

halt

// post: (1) Instance==sortInput value, but with approx.
linked list of at LEAST TWO C-strings.

// pre: Instance==the addr. of the first node of a
dnode * Spltt (dnode * Instance)

C-strings will be compared using strcmp() from <cstring>

dnode is a linked-list node type; each dnode's data is the address of a C-string.

Mergesort explained using Pre/Postconditions:
Non-Functional Requirement: Use the Merge Sort Algorithm.

```c
int strcmp (__order, `order.

post: the return value of thearrass sorted into
      the linked list of the original C-strings.
non-empty) linked list of C-strings.

pre: Instance of the address of a
```
(sorted lists computed by recursive MERGE, SORT calls)

MERGE sorts ALL lists. MERGE calls MERGE to combine the
already sorted lists.

MERGE is intended ONLY for 2 already sorted lists.

str_cmp(order)

post: The return value is the address of the first node of
the linked list of the original C-strings sorted into

pre: list1 and list2 both equal addresses of first nodes of

dnode * merge (dnode * list1, dnode * list2);

Very Important Difference in PRECONDITION
Subpr1=SPLIT(Instance); Subpr2=Instance;
I don't care about ANYTHING ELSE!

What is the result of executing this line of code?

```
dnode * Ans1 = MergeSort ( Subpr1 ) ;
```

And what is the result of executing

```
dnode * Ans1 = MergeSort ( Subpr1 ) ;
```
What is the result of executing this line of code?

`node * Ans1 = MergeSort( Subpr1 );`
???

MergeSort(Subpr2);

Ans2 =

What is the result of executing:

Subpr2

Caterpillar

NULL
The result of the recursive call is to make the returned list be SORTED properly! The result of Subpr2 = MergeSort(Subpr2)!

**Property:**

- Cat 
- Dog
- Ant
- Zebra
- NULL
Let's complete the job: First, redraw the diagram.

```c
Ans1 = MergeSort(Subpr1)
Ans2 = MergeSort(Subpr2)
```
Merge
Combination operation:

```
MergeSort(Subpr1)
Ans1 =
```

```
MergeSort(Subpr2)
Ans2 =
```

These two sorted lists...

```
dnode * Ans1 = MergeSort(Subpr1);

MergeSort(Subpr2);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
dnode * Ans2 = MergeSort(Subpr2);
```

```
dnode * Ans1 = MergeSort(Subpr1);
```

```
```
Move all remaining list or list entries to end of outputList;

// End of while loop

{ }

move the first entry from list 2 to the end of outputList;

else

{ }

move the first entry from list 1 to the end of outputList;

(there is a first entry in list 2)

if (there is a first entry in list 1)

while (there is a first entry in list 1)

OutputList = empty list;

merge (sorted list 1, sorted list 2) algorithm:
{ 
    Solution = result from COMBINING CALCULATION (Ansl, Ansl2, etc.);
    answer-type Ansl = SolutionFun(Subproblem1);
    answer-type Ansl2 = SolutionFun(Subproblem2);

    // RECURSE one, two or more times:
    Input-type Subproblem1, Subproblem2, etc.;
    Solution = result from DIRECT CALCULATION (Instance);
}

else
{
    Solution = result from DIRECT CALCULATION (Instance);

    if (Instance is small enough)
    {
        answer-type Solution;
        answer-type SolutionFun(Input-type Instance);
    }
}

Divide and Conquer pattern or paradigm
A rooted tree is a structure of nodes and arcs (pairs of nodes) that has:

(a) One root node. (and)
(b) Zero or more rooted trees, with no nodes or arcs in common with each other.
(c) One arc from this tree’s root to the root of each of the trees specified under.

What is a tree?
XXX employees.

smartness with the complex $C++$ precedence/associativity rules, FIRE that
obvious. (2) If a programmer you are supervising tries to show off his/her
2 Practical Rules: (1) If it's doubtful or subtle, USE PARENTHESES to make it

$$
\begin{align*}
35 &= 3 \times 7 = 3 \times (4 + 4) \\
23 &= 3 + 20 + 3 = (3 \times 4) + 3
\end{align*}
$$

not precedence than addition from elementary school:

must memorize or look up. They begin with "multiplication has higher
Rather complicated operator precedence and associativity rules people

How do you know which operator is evaluated first?

$A = -C - D * E + F * G$

Not fully parenthesized:

expressions.

To make learning these ideas easier, we will start with fully parenthesized
The top level operation is executed last! Why MUST it be assigned to $A$? It uses the results of the all previous operations!

7. Assign it also to $A$.

6. Assign the last sum to $B$.

5. Add subtraction's result to this last product.


3. Subtract that from $C$, remember result.

2. Multiply old value of $E$ by $D$.

1. Increment $E$ first.

means:

$$(((C * (P + (((E + D) - 4) + F) + G)) + C = B))$$

fully parenthesized:

$$A = B = C + D * E + F * G$$
Example of an expression and its Parse Tree

(A = ((B - (D * (E++))) + (F * G))))
top level operator is multiplication ( )

( F * G )

identifier

identifier
The top level operator is subtraction. The expression is:

\[(C - (D \times (E + +))\]

The diagram shows the structure of the expression with the top level operator indicated at each level.
(q) are substrings of the expression.

(3) Any operator and operands under

overlapped (and) no

or more expressions as operands (no

or has a top level operator, except-

- Either is an identifier or constant,

An expression

under (q).

the root of each of the trees specified

(c) One arc from this tree's root to

other or the root. (and)

no nodes or arcs in common with each

(b) Zero or more rooted trees, with

(a) One root node. (and)

A tree has:

definitions

these

expression, FIT these

YOUR JOB: Check that these examples, purporting

University at Albany Computer Science Dept.
Tree def: clause (p) is OK?
other trees, nothing in common.

Clause (c) is OK too!

identifier

(top level operator is increment (++))

identifier

(E++)

(top level operator is multiplication (/*))

(D*(E++))

(top level operator is subtraction (−))

(C−(D×(E++)))
Clause (c) is OK.

The operands are substituting:

Expression del. Clause (b) is OK.

Not overlapping.

Expressions as operands.

Is 1 or more

Which

Identifier

Identifier

Identifier

D

(E++)

top level operator is increment (++)

(E++)

Identifier

Identifier

Identifier

C

Has an operator:

D

(E++)

top level operator is multiplication (∗)

(E++)

D

(E++)

top level operator is increment (++)

Expression del.

Expression del.

Substring (c—D ∗ (E++) )

Substring (− (D ∗ (E++) )

Expression del.

Expression del.
and return its result.

(3) Combine the results from (2) using the meaning of the operator to compute

of the operands. (Only one cell for a unary operator.)

(2) \textbf{RECURSIVELY} Call \texttt{Evaluate(L)}, \texttt{Evaluate(R)} for each of the trees

identifier. So, return it or its value.

(1) If \( L \) is just one node only, then the expression must be a constant or

\begin{align*}
\texttt{Evaluate(ParseTree(L))}
\end{align*}

The following recursive algorithm evaluates an expression when given its parse

The tree of an expression is called the expression's \textbf{Parse Tree}.

The \textbf{Parse Tree} expression is called the expression's \textbf{Parsing expression is called the expression's \textbf{Parsing expression}}. The \textbf{Parse Tree} represents the expression's structure with absolute clarity.

The tree of an expression represents the expression's structure with

University at Albany Computer Science Dept.
recursive function.

LAB 4 is to OBSERVE the stack of ACTIVATION RECORDS during the run of a

(Insert into a stack is called push, delete from a stack is called pop.

and non-recursive.

other data relevant to all C/C++ function calls and returns, both recursive
runs C/C++ programs. Implementing and organizing local variables and

3. (The "run-time stack" of activation records, internal to the system when it

2. Storing and organizing intermediate results when evaluating expressions.

1. Figuring which pairs of parentheses MATCH in a correctly nested

3 uses for stacks:

ONLY ONE END (called the top).

that access, insertion and deletion are permitted at

What is a stack? A stack is a sequence that is restricted so