University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Dynamic Allocation Think-Pair-Share

Moving Data Structures Requires Flattening

As you know,

- **pointers** are memory addresses
- and **don't mean anything**
  - **on other computers, nor**
  - **in a later execution** of the same program.

When a program wants

- **to save a data structure** to a file,
- or **to send a data structure to another computer**,
- it must **flatten the structure**.

Flattening Means Packing into an Array of Bytes

To **flatten a data structure**,

- all **pointers must be removed**
- and the **data packed into a contiguous array of bytes**
- in a way that **allows the data structure to be rebuilt (unflattened)**.

Let's do an example of unflattening …

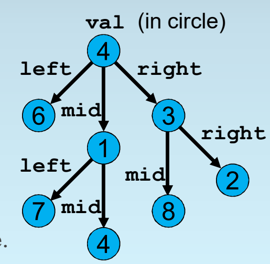
…as a think-pair-share.

But first, we'll do flattening together.

Example: Flatten the Tree Shown Here

The node structure for the tree to the right:

```
struct node_t {
    node_t* left;
    node_t* mid;
    node_t* right;
    int32_t val;
};
```



Flattening can be done in any order. Let's use the order in the structure.

Write a Recursive Function to Flatten a Tree

Let's write a function to **flatten such a tree**
- **into an array of integers**.
- **For NULL** subtrees, we **use** the symbolic constant **ABSENT**.

```
int32_t pack_tree (int32_t ar[],
    int32_t len, int32_t pos,
    node_t* root);
```

pos is the current writing position (starts at 0)

The function returns the final length written or -1 on failure (array too short to fit the tree).

Stopping Condition: Reached an Empty Subtree

We'll write the function recursively.

First, we check for **NULL**:

```
if (NULL == root) {
    if (len <= pos) {
        return -1;
    }
    ar[pos] = ABSENT;
    return (pos + 1);
}
```

Enough space to write **ABSENT**?

Add **ABSENT** to end of array.

Indicate that another space has been used.

Pack the Three Subtrees Recursively

Next, we write the three subtrees recursively.

On failure, we also fail.

```
if (-1 == (pos = pack_tree

    -1 =

    -1 =

    retu
}
```

This code is a little tricky.

First, the leap of faith:
pack_tree writes a tree into an array.
It works.
We haven't finished writing it yet.
But we have to assume that it works.
If it fails, it returns -1.

Pack the Three Subtrees Recursively

Next, we write the three subtrees recursively.

On failure, we also fail.

```
if (-1 == (pos = pack_tree
        (ar, len, pos, root->left)) ||
```

Check for failure.

Return value gives the new array position for writing.

Pass current array position for writing.

On failure, logical OR stops evaluating!

Pack the Three Subtrees Recursively

Next, we write the three subtrees recursively.
On failure, we also fail.

```
if (-1 == (pos = pack_tree
        (ar, len, pos, root->left)) ||
    -1 == (pos = pack_tree
        (ar, len, pos, root->mid)) ||
    -1 == (pos = pack_tree
        (ar, len, pos, root->right))) {
    return -1;
}
```

Only called if first call succeeds.

In which case, position is that returned from the first call.

Pack the Three Subtrees Recursively

Next, we write the three subtrees recursively.
On failure, we also fail.

```
if (-1 == (pos = pack_tree
        (ar, len, pos, root->left)) ||
    -1 == (pos = pack_tree
        (ar, len, pos, root->mid)) ||
    -1 == (pos = pack_tree
        (ar, len, pos, root->right))) {
    return -1;
}
```

Control flow and data between second call and third call is exactly the same.

Finally, Write the Node's Value

```
if (len <= pos) {
    return -1;
}
ar[pos] = root->val;
return (pos + 1);
```

Enough space to write value?

Add value to end of array.

Indicate that another space has been used.

Time for Another Think-Pair-Share

As before, let's do a group exercise in lecture.

The process:

1. I give you a problem.
2. You form groups of 3-4 people.
3. Talk about ways to solve the problem.
4. Once enough of the groups have finished, one group volunteers to share their answer.
5. We go over the group's answer together.

Your Task: Unflatten a Tree

The task: recursively unflatten
- array **ar** (left, mid, right, val order)
- into a dynamically-allocated tree of nodes.
- **pos** initially points to copy of array length, so read array from right to left
- Non-existent children appear as ABSENT (symbolic name) in the array.

```
node_t* build_tree (int32_t const ar[],
    int32_t* pos);
```

If anything goes wrong, use (and write) recursive

```
void free_tree (node_t* root);
```

to free a node and all children, and set **(*pos)** < 0.