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

ECE 220: Computer Systems & Programming

Pyramid Tree I/O Example

Use Pyramid Trees to Write Output and Input Examples

Let's do an I/O example using pyramid trees.

Here's what we'll do:

- write a tree as **ASCII**
- write a tree as binary
- compare the two files, and
- rebuild a tree from the binary file.

Then, as a think-pair-share, you can rebuild a tree from the **ASCII** file.

Pyramid Tree Nodes Consist of Four Fields

Recall the pyramid tree node structure:

```
struct pyr_node_t {
    int32_t x;
    int32_t y_left;
    int32_t y_right;
    int32_t id;
};
```

x and y splitters (internal nodes) or position (leaf nodes)

graph vertex array index (leaf nodes)

Pyramid Tree is a Number and an Array of Nodes

And the pyramid tree:

```
struct pyr_tree_t {
    int32_t     n_nodes;
    pyr_node_t* node;
};
```

number of nodes in pyramid tree

array of nodes

Write the Number of Nodes First in the File

How should we write the pyramid tree?

Start by writing the number of nodes.

Why?

When we read the tree,
- we need to **dynamically allocate**
- the **array of nodes**.
- But, to do so, we **need to know the size**.

Write the size first to make that task easier.

Given Tree and File Name, Try to Write ASCII File

Let's start the code:

```
int32_t write_pyr_tree_ASCII
    (pyr_tree_t* p, const char* fname)
{
    FILE* out;
    if (NULL == (out =
                 fopen (fname, "w"))) {
        return 0;
    }
    fprintf (out, "%d\n", p->n_nodes);
```

1 on success, 0 on failure

the file name

the tree

Open the File and Write the Number of Nodes

Let's start the code:

```
int32_t write_pyr_tree_ASCII
    (pyr_tree_t* p, const char* fname)
{
    FILE* out;
    if (NULL == (out =
                 fopen (fname, "w"))) {
        return 0;
    }
    fprintf (out, "%d\n", p->n_nodes);
```

new stream

Open file for writing.

Failed? Give up.

Print number of nodes to stream.

Write Contents of Nodes Distinctly for Internal/Leaf

What about the nodes?

For **internal nodes**, the **id field means nothing**.

So we can **write a node's contents as follows**:

<x> <y_left> <y_right>

For **leaf nodes**,
- **all fields are meaningful**,
- but, if we have the graph,
- we can find x and y position using **id**.

So, **for each leaf node**, we can **write**:

<id>

Use Equation for Identifying Leaf Nodes to Find the First

What's the index of the first leaf node?

Remember that

- **node N is a leaf node iff**

$$4N + 1 \geq \text{n\_nodes}, \text{ so}$$

$$4N \geq \text{n\_nodes} - 1$$

Dividing by 4, we obtain

$$N \geq \frac{\text{n\_nodes} - 1}{4}$$

Calculate the Index of the First Leaf Node L

$$N \geq \frac{\text{n\_nodes} - 1}{4}$$

The smallest such **N** is the first leaf node, **L**.

Since **L** is an integer, we round up,

- but integer arithmetic in **C** rounds toward zero,
- so we obtain:

$$L = \left\lceil \frac{\text{n\_nodes} - 1}{4} \right\rceil = \left\lfloor \frac{\text{n\_nodes} - 1 + 3}{4} \right\rfloor = \left\lfloor \frac{\text{n\_nodes} + 2}{4} \right\rfloor.$$

Back to the Code: Calculate the First Leaf's Index

```
int32_t first_leaf;
int32_t i;

first_leaf = (p->n_nodes + 2) / 4;
for (i = 0; first_leaf > i; i++) {
    fprintf (out, "%d %d %d\n",
             p->node[i].x,
             p->node[i].y_left,
             p->node[i].y_right);
}
```

Calculate first leaf node's index.

Loop Over All Internal Nodes and Print Each

```
int32_t first_leaf;
int32_t i;

first_leaf = (p->n_nodes + 2) / 4;
for (i = 0; first_leaf > i; i++) {
    fprintf (out, "%d %d %d\n",
             p->node[i].x,
             p->node[i].y_left,
             p->node[i].y_right);
}
```

Loop over all internal nodes.

Print x and y splitters.

Loop Over All Leaf Nodes and Print **id** Field

```
// After last loop, i is first_leaf.
for ( ; p->n_nodes > i; i++) {
    fprintf (out, "%d\n",
             p->node[i].id);
}
return (0 == fclose (out));
```

Loop over all leaf nodes.

Print **id** field.

Close file and return 0 or 1.

In Binary Version, First Open the File as a Stream

What about the binary version?

```
int32_t write_pyr_tree_binary
    (pyr_tree_t* p, const char* fname)
{
    FILE* out;
    if (NULL == (out =
                 fopen (fname, "w"))) {
        return 0;
    }
```

First part is identical to the ASCII version.

Write Number of Nodes Followed by Node Array

```
int32_t rval =
    (1 == fwrite (&p->n_nodes,
     sizeof (p->n_nodes), 1, out) &&
     p->n_nodes == fwrite
      (p->node, sizeof (p->node[0]),
       p->n_nodes, out));
fclose (out);
return rval;
```

Write **n_nodes** to output file.

Write node array to output file.

Close Output Stream and Return Success or Failure

```
int32_t rval =
    (1 == fwrite (&p->n_nodes,
     sizeof (p->n_nodes), 1, out) &&
     p->n_nodes == fwrite
      (p->node, sizeof (p->node[0]),
       p->n_nodes, out));
fclose (out);
return rval;
```

Return success if both writes succeed.

Close the output stream.

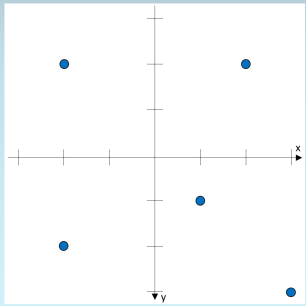
Return success or failure.

Use a Small Graph as an Example

Here's a small graph with **5 vertices** and **no edges**.

The pyramid tree has **7 nodes**.

$L = \left\lfloor \frac{7+2}{4} \right\rfloor = 2.$

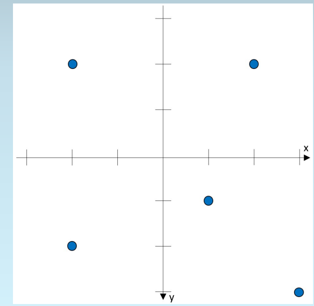


Look at the Output for a Small Graph

The ASCII file for the pyramid tree is:

```
7
1 1 0
-1 -2 1
2
0
3
1
4
```

total: 26 bytes



Look at the Output for a Small Graph

The ASCII file for the pyramid tree is:

```
7
1 1 0
-1 -2 1
2
0
3
1
4
```

total: 26 bytes

The binary file for the pyramid tree is:
- **4B** for **`n_nodes`**
- **16B** per node
- **7** nodes

total: 116 bytes

ASCII is Smaller Because We Left Out Unnecessary Bits

The results are similar for the graph of the streets in the Champaign-Urbana area:
- **570,555B for ASCII**, and
- **942,164B for binary**.

Why is the binary file larger?

We saved a lot of space by not writing everything.
- If we had written all four fields
- for all nodes in **ASCII**,
- the result is over **1.5MB**.

And most numbers are small.

Could have done the same with the binary file.

Read and Build a Pyramid Tree from a File

Now, let's **reconstruct a pyramid tree from a binary file.**

```
pyr_tree_t* read_pyr_tree_binary
     (const char* fname)
{
     FILE*          in;
     pyr_tree_t* p;
     int32_t        count;
```

Returns new tree or NULL.

file name

input stream

new pyramid tree

number of nodes in file

Open File for Reading, Then Read Number of Nodes

```
if (NULL == (in =
             fopen (fname, "r")) ||
    1 != fread (&count,
             sizeof (count), 1, in)) {
    if (NULL != in) {
        fclose (in);
    }
    return 0;
}
```

Open file for reading.

If file open succeeds, read number of nodes in file.

On Failure, Try to Close Stream, Then Return Failure

```
if (NULL == (in =
             fopen (fname, "r")) ||
    1 != fread (&count,
             sizeof (count), 1, in)) {
    if (NULL != in) {
        fclose (in);
    }
    return 0;
}
```

If either fails, try to close stream, then return failure.

Allocate Space for Pyramid Tree and Node Array

Allocate space for pyramid tree.

```
if (NULL ==
    (p = malloc (sizeof (*p))) ||
    NULL == (p->node = malloc
   (count * sizeof (p->node[0])))) {
    if (NULL != p) { free (p); }
    fclose (in);
    return NULL;
}
```

Allocate space for node array.

On Failure, Free Tree and Close Stream, Then Return

```
if (NULL ==
     (p = malloc (sizeof (*p))) ||
     NULL == (p->node = malloc
    (count * sizeof (p->node[0])))) {
     if (NULL != p) { free (p); }
     fclose (in);
     return NULL;
}
```

If either fails, try to free tree, close stream, then return failure.

Read Node Array from Input Stream

Write number of nodes into pyramid tree.

```
p->n_nodes = count;
if (p->n_nodes != fread
      (p->node, sizeof (p->node[0]),
       p->n_nodes, in)) {
    free_pyramid_tree (p);
    fclose (in);
    return NULL;
}
```

Read node array from stream.

If node array read fails, free tree, close stream, and return failure.

Clean Up and Return the New Pyramid Tree

Discard the return value (explicit).

```
(void) fclose (in);
return p;
```

Close the input stream

Return the new pyramid tree.

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: Rebuild a Pyramid Tree from an ASCII File

The task: read a file and build a pyramid tree
- using information written into file earlier:
- internal nodes: **`x y_left y_right`**
- leaf nodes: **`id`**

```
pyr_tree_t* read_pyr_tree_ASCII
    (const char* fname, graph_t* g);
```

Return a new tree on success,
or **`NULL`** on failure.

(Recall: use **`g->vertex[id].x`** and **`g->vertex[id].y`**.)