Lists and Trees in C

Alan L. Cox

alc@cs.rice.edu

Lists, Trees, …

Not built-in

- However, many libraries
  - e.g., `#include <sys/queue.h>`

Combination of structs, unions, enums, & pointers

- Adapts ideas from COMP 210/212
- Verbose, but straight-forward
- Some standard shortcuts

List Example

A list of `int` is either

- Empty or
- an `int` and a list of `int`

Use a union to allow either case:

```
union {
  empty
  ACons
};
```

Tag the Union

Differentiate between cases

```
enum ListTag {LIST_EMPTY, LIST_CONS};
struct ListElt {
  enum ListTag tag;
  union {
    empty
    ACons
  } elt;
};
```

Must name struct field!

Empty Case

Use `void` as a placeholder for the empty case

- **No value has type `void`**
- **`void` represents no information other than its presense**

```
enum ListTag {LIST_EMPTY, LIST_CONS};
struct ListElt {
  enum ListTag tag;
  union {
    void empty;
    ACons
  } elt;
};
```

Cons Case

Use a struct to hold an int and a list of int

- **First member is the `int`**
- **Second member is the rest of the list**

```
enum ListTag {LIST_EMPTY, LIST_CONS};
struct ListElt {
  enum ListTag tag;
  union {
    void empty;
    struct {
      int data;
      list
    } cons;
  } elt;
};
```

Must name union field!

The Rest of the List

Just another element…

```
enum ListTag {LIST_EMPTY, LIST_CONS};
struct ListElt {
  enum ListTag tag;
  union {
    void empty;
    struct {
      int data;
      struct ListElt *next;
    } cons;
  } elt;
};
```

Problem: how big is this?

struct ListElt not yet defined!

Solution: pointers are all the same size

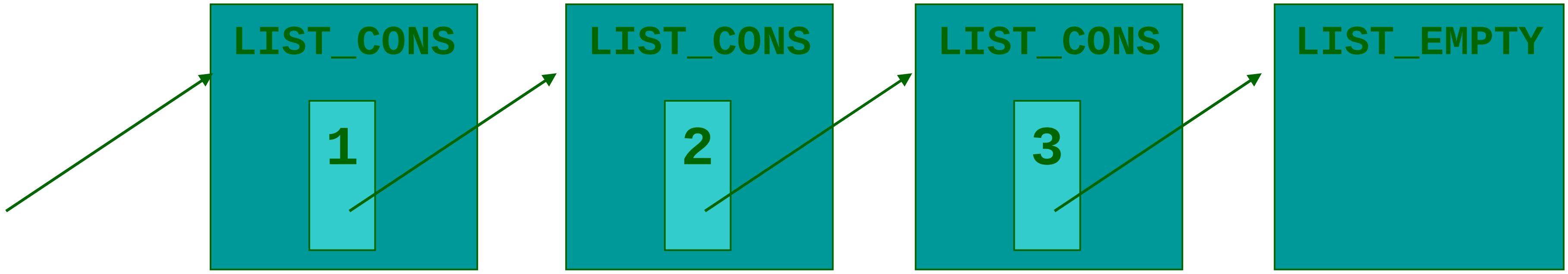
Always use pointers for recursive structures

Simplify the Type Name

```
enum ListTag {LIST_EMPTY, LIST_CONS};
struct ListElt {
  enum ListTag tag;
  union {
    void empty;
    struct {
      int data;
      struct ListElt *next;
    } cons;
  } elt;
};

typedef struct ListElt *List;
```

List Example: Picturing Lists



List Example: Creating Lists

```
List make_empty(void)
{
   List list = Malloc(sizeof(struct ListElt));
   list->tag = LIST_EMPTY;
   return (list);
}

List make_cons(int data, List next)
{
   List list = Malloc(sizeof(struct ListElt));
   list->tag = LIST_CONS;
   list->elt.cons.data = data;
   list->elt.cons.next = next;
   return (list);
}
```

Reminder: Check return value for **`NULL`**

`list->elt` is equivalent to `(*list).elt`

List Example: Accessing Lists

```
List  list;
…
if (list == NULL) {
   fprintf(stderr,“Program error: Invalid list.\n”);
   exit(1);
}
switch (list->tag) {
case LIST_EMPTY:
   …;
   break;
case LIST_CONS:
   … list->elt.cons.data … list->elt.cons.next …;
   break;
default:
   fprintf(stderr,“Program error: Invalid list tag.\n”);
   exit(1);
}
```

NULL pointer isn't valid data of this type (Remember this!) Must check, because you can't dereference a NULL pointer!

Always check for errors (Compiler might be able to remove checks for efficiency)

Pointers to Data Structures

<u>Always</u> refer to complex data via pointers!!!

Why?

- **Correctness: Avoids most common mistakes, such as…**
- **Efficiency: Copying word-sized pointers better than copying large complex data**
- **Familiarity: This is what Java, Scheme, etc., do implicitly**

```
struct ListElt *foo(…)
{
   struct ListElt  list;
   …;
   return (&list);
}
```

```
typedef  struct ListElt *List;
```

Simplifying

Previous steps work for any inductive data structure

But, resulting type is very verbose

Often, can simplify…

Eliminate void Data

Union tag is sufficient information

Remove single-case union

```
enum ListTag {LIST_EMPTY,
              LIST_CONS};
struct ListElt {
  enum ListTag tag;
  union {
    void empty;
    struct {
      int data;
      struct ListElt *next;
    } cons;
  } elt;
};
```

```
enum ListTag {LIST_EMPTY,
              LIST_CONS};
struct ListElt {
  enum ListTag tag;
  struct {
    int data;
    struct ListElt *next;
  } cons;
};
```

Use NULL for void Case

NULL pointer indicates "empty" case

- Must always refer to elements with pointers
- Eliminates need for ListTag

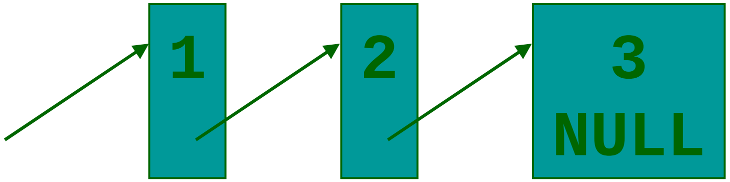
Collapse nested structures

```
enum ListTag {LIST_EMPTY,
              LIST_CONS};
struct ListElt {
  enum ListTag tag;
  struct {
    int data;
    struct ListElt *next;
  } cons;
};
```

```
struct ListElt {
  int data;
  struct ListElt *next;
};
```

This is the typical C list definition

Simplified List Example



Simplified List Example: Creating & Accessing

Creating data:

```
List make_empty(void)
{
   return (NULL);
}

List make_cons(int data, List next)
{
   List list = Malloc(sizeof(struct ListElt));
   list->data = data;
   list->next = next;
   return (list);
}
```

Accessing data:

```
List list = …;

if (list == NULL) {
   …
} else {
   … list->data … list->next …
}
```

Binary Tree Example 1

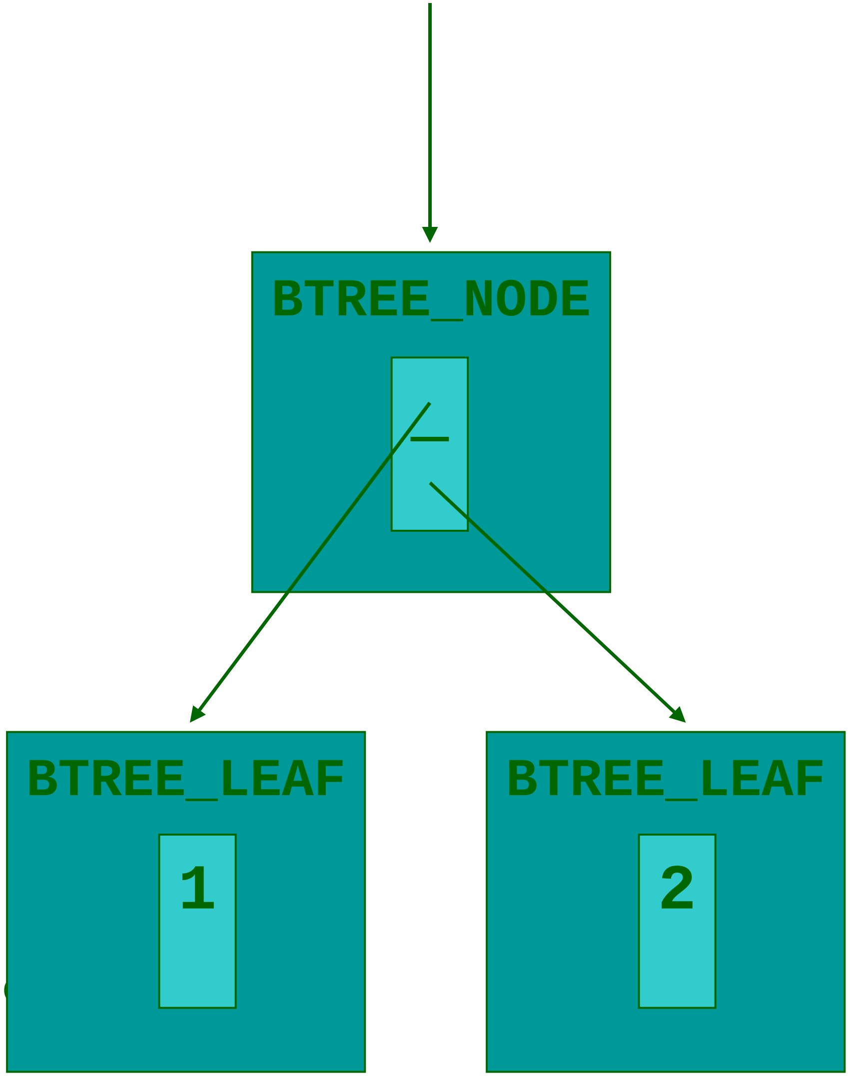
Use same steps to define these binary trees:

A binary tree of `int` is either

- **an `int` or**
- **a binary tree of `int` and another binary tree of `int`**

Binary Tree Example 1

```
enum BTreeTag {BTREE_LEAF, BTREE_NODE};
struct BTreeElt {
   enum BTreeTag tag;
   union {
      int  leaf;
      struct {
         struct BTreeElt *left;
         struct BTreeElt *right;
      } node;
   } elt;
};
typedef struct BTreeElt *BTree;
```



Simplification steps aren't relevant
No void case: NULL is not a valid value

Binary Tree Example 2

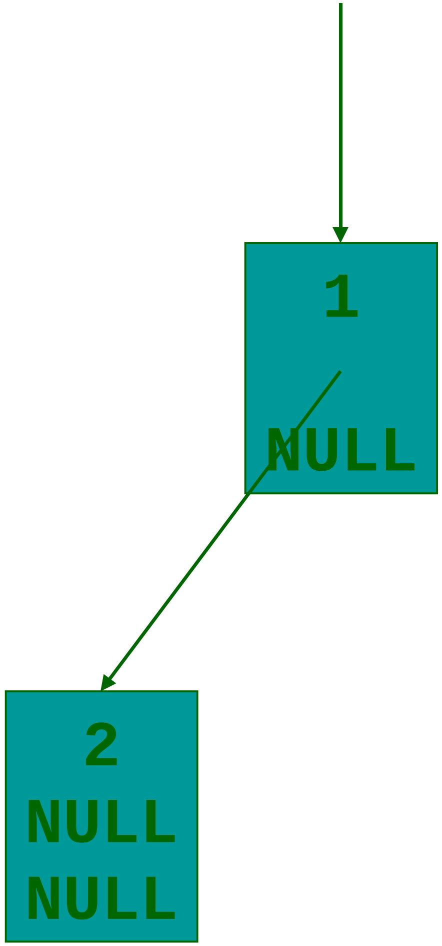
Use same steps to define these binary trees:

A binary tree of `int` is either

- `empty` **or**
- **an `int`, a binary tree of `int`, and another binary tree of `int`**

Binary Tree Example 2

```
struct BTreeElt {
   int              value;
   struct BTreeElt *left;
   struct BTreeElt *right;
};
typedef struct BTreeElt *BTree;
```



Simplifications steps are relevant

`NULL` represents empty tree

Mono- & Polymorphism

C has a *monomorphic* type system

- **Our `List` & `BTree` definitions only allowed `ints`**
- **Need separate definitions for types of same structure, but containing different types of elements**
- **There are ways around this, using …**
  - **Macros, e.g.,** `#include <sys/queue.h>`
  - `void *`
    - **Reduces the effectiveness of type-checking**

```
struct ListElt {
  void *data;
  struct ListElt *next;
};
```

Most modern languages have some form of *polymorphism*

Next Time

Brief Tour of x86-64 Assembly Language