CSE 326: Data Structures
Extra Slides on Nested Lists

Henry Kautz

Winter 2002

These Slides Contain

- Example of a polymorphic node type.
- Method for representing a tree as a list containing the root followed by pointers to each child.
- Simpler LISP-like method for representing a tree as a list contain the root followed by the children (*not* pointers to the children).
- The LISP-like method, but using a generic Node template and a polymorphic object type.
  - See file "poly.cpp" on the course web page for a file containing code for this final version.

Polymorphic* Node

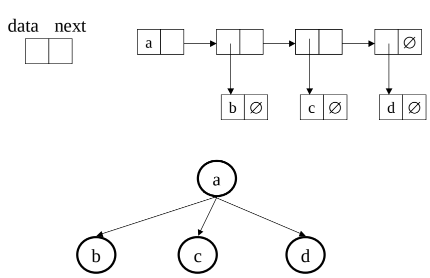
**polymorphic: able to contain different types*

```
class node {
public: enum Tag { I, P };
private:
 union { int i; node * p; };
 Tag tag;
 void check(Tag t){ if (tag!=t) error();}
 node * next;
public:
 Tag get_tag() { return tag; }
 int & ival() { check(I); return i; }
 node * & pval() { check(P); return p; }
```

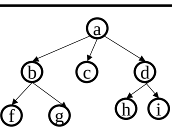
Creating and Setting Nodes

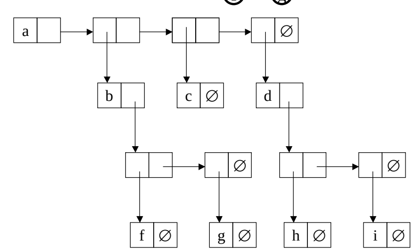
```
class node {
...
public:
 // Creating a new node
 node(int ii) { i=ii; tag=I; }
 node(node * pp) { p=pp; tag=P; }
 // Changing the value in a node
 void set(int ii) { i=ii; tag=I; }
 void set(node * pp) { p=pp; tag=P; }
};
```

Method 1: Nested List Implementation of a Tree



How To Represent?



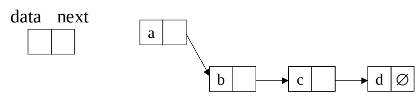


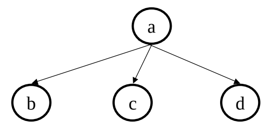
Recursive Preorder for Method 1 Nested List Implementation

```
void print_preorder ( Node * n)
{     Node * np;

      if ( n == NULL ) return;
      cout << (n -> ival()); // non-pointer data
      np = n -> next;
      while (np != NULL) {
            print_preorder ( np->pval() );
            np = np->next;
      }
}
```

"LISP" Nested List Implementation of a Tree

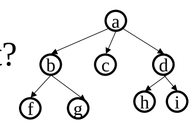


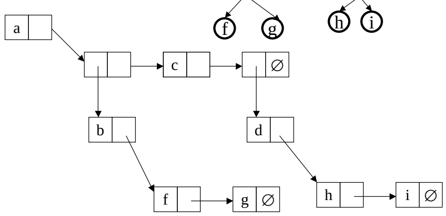


-uses fewer nodes

-LISP convention

How To Represent?





Recursive Preorder for "LISP" Nested List Implementation

```
void print_preorder ( Node * n)
{
  while (n != NULL){
      if ( n->get_tag()==I ) cout << n->ival();
      else // must be the case that get_tag()==P
            print_preorder( n->pval() );
      n = n -> next;
}
```

Using Distinct Node and Polymorphic Objects

```
template <class t> struct node; //forward declaration

class poly {
public: enum Tag { I, P };
private:
 union { int i; node<poly> * p; };
 Tag tag;
 void check(Tag t){ if (tag!=t) error("bad");}
public:
 Tag get_tag() { return tag; }
 int & ival() { check(I); return i; }
 node<poly> * & pval() { check(P); return p; }
```

Using Distinct, ... continued

```
public:
 // Creating a new poly
 poly() { i=0; tag=I; }
 poly(int ii) { i=ii; tag=I; }
 poly(node<poly> * pp) { p=pp; tag=P; }
 // Changing the value in a poly
 void set(int ii) { i=ii; tag=I; }
 void set(node<poly> * pp) { p=pp; tag=P; }
 void print_preorder(void);
};

template <class t> struct node {
  t data;
  node<t>* next; };
```

Recursive Preorder for Distinct Node/Poly Implementation

```
void poly::print_preorder (void)
{
  if (get_tag()==I) cout << ival() << " ";
  else { // must be pointer to a node
    node<poly> * np = pval();
    while (np != NULL){
      (np->data).print_preorder();
      np = np->next;
    }
  }
}
```

Other Choices

- Use an explicit List class as well as a Node class or structure
- pval() then is a List, rather than a pointer to Node
- print_preorder() or other routines that traverse the tree would need some way to efficiently step through the nodes in the list.
  - I.e., don't actually destroy the list using Pop()
  - You could let the List() give you it's first node, or you could define a list iterator type as described in the textbook.