General Trees

Chapter 7

Well, “non-binary” trees anyway.

More formally...

- A **tree**, $T$
  - is a finite set of one or more nodes
  - such that there is one designated node $r$ called the root of $T$
  - and the remaining nodes in $(T \setminus \{r\})$ are partitioned into $n \geq 0$ disjoint subsets $T_1, T_2, \ldots, T_k$
  - each of which is a tree
  - and whose roots $r_1, r_2, \ldots, r_k$, respectively, are children of $r$

General Trees

- General trees are similar to binary trees, except that there is no restriction on the number of children that any node may have.

N-ary Trees

- An **n-ary tree** is a generalization of a binary tree, where each node can have no more than $n$ children.
- Since the maximum number of children for any node is known, each parent node can **point directly** to **each of its children** -- rather than requiring a linked list.
N-ary Trees

- This results in a faster search time (if you know which child you want).
- The disadvantage of this approach is that extra space reserved in each node for \( n \) child pointers, many of which may not be used.

General Trees: Example

- An \( n \)-ary tree with \( n = 3 \)
- Pointer-based implementation of the \( n \)-ary tree

General Trees: Example (Cont’d.)

- Binary tree with the pointer structure of the preceding general tree
- Pointer-based implementation of the general tree

Tree ADT (Java)

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - boolean isLeaf()
  - boolean isExternal(p)
  - boolean isRoot(p)
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update method:
  - object replace(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

C++ ADT

```cpp
// C++ ADT

class GTNode {
public:
    GTNode (const ELEM); // constructor
    ~GTNode(); // destructor
    ELEM value(); // return node’s value
    bool isLeaf(); // TRUE if is a leaf
    GTNode* parent(); // return parent
    GTNode* leftmost_child(); // return first child
    GTNode* rightmost_sibling(); // return right sibling
    void setValue(ELEM); // set node’s value
    void insert_first(GTNode* n); // insert first child
    void insert_next(GTNode* n); // insert right sibling
    void remove_first(); // remove first child
    void remove_next(); // remove right sibling
};
```
C++ ADT

Class GenTree {
    public:
        GenTree(); // constructor
        ~GenTree(); // destructor
        void clear(); // free nodes
        GTNode* root(); // return root
        void newroot(ELEM, GTNode*, GTNode*); // combine
    }

General Tree Traversal

Algorithm Print (GTNode rt) // preorder traversal from root
Input: a general tree node
Output: none – information printed to screen

GTNode temp
  if (rt is a leaf)
    output "Leaf: "
  else
    output "Internal: ";
    output value stored in node
    temp = leftmost_child of rt
    while (temp is not NULL)
        Print (temp) // note recursive call
        temp = right_sibling of temp

Preorder Traversal:
1) process root
2) recursively process children from left to right

A B C D E F G H I J K L

Postorder Traversal:
1) no node is processed until all of
   its children have been processed,
   recursively, left to right
2) process root

F P M N O G H B I C D J K L E A

Inorder Traversal:
by definition, none

A B C D E F G H I J K L

Preorder Traversal:
1) process root
2) recursively process children from left to right

A B F G M P N O H C I D E J K L
Parent Pointer Implementation

Implementations

Common ones, plus make up your own!

Lists of children

Leftmost Child/Right Sibling

Linked Implementations
Linked Implementations

Left child/right sibling representation essentially stores a binary tree. Use this process to convert any general tree to a binary tree.

A forest is a collection of one or more general trees.

Sequential Implementations

List node values in the order they would be visited by a preorder traversal.

Saves space, but allows only sequential access.

Need to retain tree structure for reconstruction.

Sequential Implementations

Example: For binary trees, use a symbol to mark null links.

\[
\text{AB/D//CEG///FH///} \\
\text{Which node was the right child of the root?}
\]

space efficient, but not time efficient
Sequential Implementations

Example: Mark nodes as leaf or internal.

A'B'/DC'E'G/F'H

no need for null pointers when both children are null

What about general trees?

- Not only must the general tree implementation indicate whether a node is a leaf or internal node, it must also indicate how many children the node has.

- Alternatively, the implementation can indicate when a node’s child list has come to an end.

- Include a special mark to indicate the end of a child list.

- All leaf nodes are followed by a “)” symbol since they have no children.

- A leaf node that is also the last child for its parent would indicate this by two or more successive “)” symbols.

Sequential Implementations

Example: For general trees, mark the end of each subtree.

RAC)D(E))BF))