Chapter 5 Trees

- Introduction
- Binary Trees
- Binary Tree Traversals
- Additional Binary Tree Operations
- Threaded Binary Trees
- Heaps
- Binary Search Trees
- Selection Trees
- Forests
- Representation of Disjoint Sets
- Counting Binary Trees

Introduction

Computer Scientist’s View

root

branches

nodes

leaves
Pedigree - An ancestor tree

Root

Dusty

Honey Bear

Brunhilde

Tansey

Gill

Tweed

Terry

Zoe

Coyote

Tweed

Crocus

Primrose

Nugget

Nous

Belle

Tree Definition

A tree is a finite set of one or more nodes such that:

- There is a specially designated node called the root.
- The remaining nodes are partitioned into \( n \geq 0 \) disjoint sets \( T_1, \ldots, T_n \), where each of these sets is a tree.

We call \( T_1, \ldots, T_n \) the sub-trees of the root.
Terminology

- A node that has subtrees is the *parent* of the roots of the subtrees, e.g., Parent (A).
- The roots of these subtrees are the *children* of the node, e.g., Children (E, F).
- Children of the same parent are *siblings*, e.g., Siblings (C, D).

Terminology (Cont’d)

- The root of this tree is node A. Root (A).
- The *degree* of a node is the number of sub-trees of the node.
- The *level* of a node: Initially letting the root be at level one
  - For all other nodes, the level is the level of the node's parent plus one.
- Nodes that have degree zero are called Leaves, e.g., K, L, F, G, M, I, J.
- The *height* or *depth* of a tree is the maximum level of any node in the tree
Representation of Trees (1)

- **List Representation**
  - The root comes first, followed by a list of sub-trees
  - Example:
    \[(A(B(E(K,L),F),C(G),D(H(M),I,J)))\]

<table>
<thead>
<tr>
<th>data</th>
<th>link 1</th>
<th>link 2</th>
<th>...</th>
<th>link n</th>
</tr>
</thead>
</table>

A node must have a varying number of link fields depending on the number of branches

---

An Example:
List Representation of a Tree

```
A
  B
    E
    K
    L
    0
  F
  0
  C
    G
    0
  D
    H
    I
    J
    0
  0
  H
    M
    0
```

---

C-C Tsai P.7

C-C Tsai P.8
Representation of Trees (2)

- **Left Child-Right Sibling Representation**
  - A degree-two tree, i.e., binary tree
  - Rotate clockwise by 45°

```
       A
      /   \
     B     C
    /     /  \
   E     F    D
  /     /    /  \
 K     L     G    H
```

```
       A
      /   \       \
     B     C     D
    /     /     / \
  E     F     G    H
 /     /  /     /\   \
K     L M     I    J
```

Binary Trees

- A **binary tree** is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the **left subtree** and the **right subtree**.
- Any tree can be transformed into a binary tree.
  - By using left child-right sibling representation.
  - The left and right subtrees are distinguished.
Abstract Data Type of Binary Trees

Structure Binary_Tree (abbreviated BinTree) is:

**Objects:** a finite set of nodes either empty or consisting of a root node, left Binary_Tree, and right Binary_Tree.

**Functions:**

For all \( bt, bt1, bt2 \in BinTree, \text{ item } \in \text{ element} \)

- Bintree Create() ::= creates an empty binary tree
- Boolean IsEmpty(bt) ::= if (bt==empty binary tree) return TRUE else return FALSE
- BinTree MakeBT(bt1, item, bt2) ::= return a binary tree whose left subtree is \( bt1 \), whose right subtree is \( bt2 \), and whose root node contains the data item
- Bintree Lchild(bt) ::= if (IsEmpty(bt)) return error else return the left subtree of \( bt \)
- element Data(bt) ::= if (IsEmpty(bt)) return error else return the data in the root node of \( bt \)
- Bintree Rchild(bt) ::= if (IsEmpty(bt)) return error else return the right subtree of \( bt \)

Special Binary Trees

- **Skewed binary tree**
  - A
  - B
  - C
  - D
  - E
  - Level 1

- **Complete binary tree**
  - A
  - B
  - C
  - D
  - E
  - F
  - G
  - H
  - I
  - Level 5
Properties of Binary Trees

- **Lemma [Maximum number of nodes]:**
  - The maximum number of nodes on level $i$ of a binary tree is $2^{i-1}$, $i \geq 1$. If $i=3$, $2^{i-1}=4$.
  - The maximum number of nodes in a binary tree of depth $k$ is $2^k - 1$, $k \geq 1$. If $i=3$, $2^{k-1}=8$.

- **Lemma [Relation between number of leaf nodes and degree-2 nodes]:**
  - For any nonempty binary tree, $T$, if $n_0$ is the number of leaf nodes and $n_2$ the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Full Binary Tree vs Complete Binary Tree

- A **full binary tree** of depth $k$ is a binary tree of depth $k$ having $2^k - 1$ nodes, $k \geq 0$.

- A binary tree with $n$ nodes and depth $k$ is **complete** iff its nodes correspond to the nodes numbered from 1 to $n$ in the full binary tree of depth $k$. 
Binary Tree Representation

Array Representation

Skewed binary tree

Complete binary tree

Lemma: If a complete binary tree with \( n \) nodes (depth = \( \lceil \log_2 n + 1 \rceil \)) is represented sequentially, then for any node with index \( i \), \( 1 \leq i \leq n \), we have:

- parent (i) is at \( \lfloor i / 2 \rfloor \), \( i \neq 1 \).
- left-child (i) is 2i, if 2i \( \leq \) n.
- right-child (i) is 2i+1, if 2i+1 \( \leq \) n.

For complete binary trees, this representation is ideal since it wastes no space. However, for the skewed tree, less than half of the array is utilized.

Waste space and insertion/deletion problems
**Linked Representation**

typedef struct node *treePointer;
typedef struct node {
  int data;
  treePointer leftChild, rightChild;
};

**Binary Tree Traversals**

- Traversing order: L, V, R
  - L: moving left
  - V: visiting the node
  - R: moving right
- Inorder Traversal: LVR
- Preorder Traversal: VLR
- Postorder Traversal: LRV
An Example of Arithmetic Expression

- Inorder Traversal (infix expression) : A / B * C * D + E
- Preorder Traversal (prefix expression) : + * * / A B C D E
- Postorder Traversal (postfix expression) : A B / C * D * E +

Inorder Tree Traversal

A recursive function starting from the root
- Move left → Visit node → Move right

typedef struct node *treePointer;
typedef struct node
{
    int data;
    treePointer leftChild, rightChild;
};

Void inorder (treePointer ptr)
{
    if (ptr)
    {
        inorder (ptr->leftChild);
        printf("%d",ptr->data);
        inorder (ptr->rightChild);
    }
}

O(n)
Example: Inorder Tree Traversal

<table>
<thead>
<tr>
<th>Call of inorder</th>
<th>Value in root</th>
<th>Action</th>
<th>inorder</th>
<th>Value in root</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td></td>
<td>11</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td></td>
<td>12</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td></td>
<td>11</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>/</td>
<td></td>
<td>13</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>NULL</td>
<td></td>
<td>14</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>printf</td>
<td>15</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>NULL</td>
<td></td>
<td>16</td>
<td>D</td>
<td>printf</td>
</tr>
<tr>
<td>9</td>
<td>B</td>
<td>printf</td>
<td>1</td>
<td>+</td>
<td>printf</td>
</tr>
<tr>
<td>10</td>
<td>NULL</td>
<td></td>
<td>17</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>printf</td>
<td>19</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

In-order Traversal: A / B * C * D + E

Preorder Tree Traversal

- A recursive function starting from the root
  - Visit node → Move left → Move right

Void preorder (treePointer ptr)
{
  if (ptr)
  {
    printf("%d", ptr->data);
    preorder (ptr->leftChild);
    preorder (ptr->rightChild);
  }
}

O(n)
**Postorder Tree Traversal**
- A recursive function starting from the root
  - Move left → Move right → Visit node

Void postorder (treePointer ptr)
{
  if (ptr)
  {
    postorder (ptr->leftChild);
    postorder (ptr->rightChild);
    printf("%d",ptr->data);
  }
}

O(n)

**Other Traversals of Binary Trees**
- **Iterative Inorder Traversal**
  - Using a stack to simulate recursion
  - Time Complexity: \(O(n)\), \(n\) is number of nodes.
- **Level Order Traversal**
  - Visiting at each new level from the left-most node to the right-most
  - Using Data Structure: Queue
Example: Iterative Inorder Traversal

| Add “+” in stack | Delete “C” & Print |
| Add “*” & Print   | Delete “E” & Print  |
| Add “/”           | Delete “D” & Print  |
| Add “A”           | Delete “+” & Print  |
| Delete “A” & Print| Add “D”             |
| Delete “/” & Print| Delete “B” & Print  |
| Add “B”           | Delete “*” & Print  |
| Delete “B” & Print| Add “C”             |
| Delete “*” & Print|                         |

Inorder Traversal: \( A / B * C * D + E \)

Algorithm of Iterative Inorder Traversal

```
void iterInorder (treePointer node)
{
    int top = -1; // initialize stack
    treePointer stack[MAX_STACK_SIZE];
    for (;;) {
        for (; node; node=node->leftChild)
            push(node); // add into stack
        node = pop(); // delete from stack
        if (!node) break; // empty stack
        printf("%d",ptr->data);
        node = node->rightChild;
    }
}
```

\( O(n) \)
Example: Level Order Traversal

<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add “+”</td>
<td>in Queue</td>
</tr>
<tr>
<td>Deleteq “+”</td>
<td></td>
</tr>
<tr>
<td>Addq “*”</td>
<td></td>
</tr>
<tr>
<td>Addq “E”</td>
<td></td>
</tr>
<tr>
<td>Deleteq “*”</td>
<td></td>
</tr>
<tr>
<td>Addq “D”</td>
<td></td>
</tr>
<tr>
<td>Deleteq “E”</td>
<td></td>
</tr>
<tr>
<td>Deleteq “*”</td>
<td></td>
</tr>
</tbody>
</table>

Addq “/”
Deleteq “/”
Addq “C”
Deleteq “D”
Addq “A”
Deleteq “/”
Addq “B”
Deleteq “C”
Deleteq “A”
Deleteq “B”

Level-order Traversal: + * E * D / C A B

Algorithm of Level Order Traversal

```c
Void levelorder (treePointer ptr)
{
    int front = rear = 0; // initialize queue
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr)  return; // empty tree
    addq(ptr);
    for (;;)
    {
        ptr = deleteQ();
        if (!ptr)
        {
            printf("%d",ptr->data);
            if (ptr->leftChild)
                addQ(ptr->leftChild);
            if (ptr->rightChild)
                addQ(ptr->rightChild);
            else break;
        }
    }
}
```
Additional Binary Tree Operations

- Copying Binary Trees
- Testing Equality of Binary Trees
- The Satisfiability Problem

Copying Binary Trees

Modified from postorder traversal program

treePointer copy(treePointer original)
{
    treePointer temp;
    if (original)
    {
        MALLOC(temp,sizeof(temp));
        temp->leftChild=copy(original->leftChild);
        temp->rightChild=copy(original->rightChild);
        temp->data=original->data;
        return temp;
    }
    return NULL;
}
Testing for Equality of Binary Trees

Equality: Two binary trees having identical topology and data are said to be equivalent.

```c
int equal(tree_pointer first, tree_pointer second)
{ /* function returns FALSE if the binary trees first and second are not equal, otherwise it returns TRUE */
    return ((!first && !second) ||
        (first && second &&
        (first->data == second->data) &&
        equal(first->left_child, second->left_child) &&
        equal(first->right_child, second->right_child)))
}
```

SAT (Satisfiability) Problem

Satisfiability problem formulation

- **Variables**: $X_1, X_2, ..., X_n$
  - Two possible values: True or False
- **Operators**: And ($\land$), Or ($\lor$), Not ($\neg$)
  - A variable is an expression.
  - If $x$ and $y$ are expressions, then $\neg x$, $x \land y$, $x \lor y$ are expressions.
  - Parentheses can be used to alter the normal order of evaluation, which is $\neg$ before $\land$ before $\lor$, ($\neg > \land > \lor$).

**Example:**

$x_1 \lor (x_2 \land \neg x_3)$

If $x_1$ and $x_3$ are False and $x_2$ is True, then the expression is True.
Example: SAT Problem

\[(x_1 \land \neg x_2) \lor (\neg x_1 \land x_3) \lor \neg x_3\]

\[\begin{array}{c}
\lor \\
\wedge \\
\neg \\
\end{array}\]

\[\begin{array}{c}
x_1 \\
\neg \\
x_2 \\
\end{array}\]

\[\begin{array}{c}
x_3 \\
\neg \\
x_1 \\
\end{array}\]

2\(^n\) possible combinations for  n variables
If \(n = 3\), \(x_1 \sim x_3\)
(t,t,t)
(t,t,f)
(t,f,t)
(t,f,f)
(f,t,t)
(f,t,f)
(f,f,t)
(f,f,f)

Node Data Structure for SAT in C

<table>
<thead>
<tr>
<th>leftChild</th>
<th>data</th>
<th>value</th>
<th>rightChild</th>
</tr>
</thead>
</table>

typedef enum {not, and, or, true, false } logical;
typedef struct node *treePointer;
typedef struct node {
treeNode   leftChild;
logical    data;
short int  value;
treeNode   rightChild;
} ;
Enumerated Algorithm for SAT

for (all $2^n$ possible combinations)
{
    generate the next combination;
    replace the variables by their values;
    evaluate root by traversing it in postorder;
    // postOrderEval(treePointer node)
    if (root->value)
    {
        printf(<combination>);
        return;
    }
}
printf(“No satisfiable combination \n”);

- The algorithm takes $O(g \cdot 2^n)$ time, $g$ is the time required to substitute the true and false values for variables and to evaluate the expression.

Postorder Evaluation Function

void postOrderEval(treePointer node)
{/* modified post order traversal to evaluate a propositional calculus tree */
    if (node)
    {
        postOrderEval(node->leftChild);
        postOrderEval(node->rightChild);
        switch(node->data)
        {
            case not: node->value=!node->rightChild->value; break;
            case and: node->value=node->rightChild->value && node->leftChild->value; break;
            case or: node->value=node->rightChild->value || node->leftChild->value; break;
            case true: node->value=TRUE; break;
            case false: node->value=FALSE; break;
        }
    }
}
Threaded Binary Trees

- In Linked Representation of Binary Tree, more null links than actual pointers. Waste!
- Threaded Binary Tree: Replace these null links with some useful “threads”.
- If ptr->leftChild is null, replace it with a pointer to the node that would be visited before ptr in an inorder traversal.
- If ptr->rightChild is null, replace it with a pointer to the node that would be visited after ptr in an inorder traversal.

Node Structure of Threaded Binary Tree

define struct threadedTree *
define struct threadedTree {
    short int leftThread;
    threadedPointer leftChild;
    char data;
    short int rightChild;
    threadedPointer rightChild;
}

- An empty threaded binary tree

<table>
<thead>
<tr>
<th>leftThread</th>
<th>leftChild</th>
<th>data</th>
<th>rightChild</th>
<th>rightThread</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>.</td>
<td></td>
<td>.</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

TRUE: thread  FALSE: child
Memory Representation of Threaded BT

Inorder Traversal of a Threaded BT

- Threads simplify inorder traversal algorithm
- For any node, ptr, in a threaded binary tree
  - If ptr -> rightThread = TRUE
    - The inorder successor of ptr = ptr -> rightChild
  - Else (Otherwise, ptr -> rightThread = FALSE)
    - Follow a path of leftChild links from the rightChild of ptr until finding a node with leftThread = TRUE
- Function insucc
  - Finds the inorder successor of any node (without using a stack)
- An easy $O(n)$ algorithm
Algorithm of Inorder Traversal of a Threaded Binary Tree

```c
void tinorder(threadedPointer tree)
{
    threadedPointer temp = tree;
    for (;;)
    {
        temp = insucc(temp);
        if (temp == tree) break;
        printf("%3c", temp->data);
    }
}
```

```c
threadedPointer insucc(threadedPointer tree)
{
    threadedPointer temp;
    temp = tree->rightChild;
    if (!tree->rightThread)
        while (!temp->leftThread)
            temp = temp->leftChild;
    return temp;
}
```

Inserting a Node into a Threaded BT

- Insert a new node as a left or right child of a parent node
- Is the original child node an empty subtree?
  - Empty child node (parent -> childThread = TRUE)
  - Non-empty child node (parent -> childThread = FALSE)

**Example:** Inserting a node as the right child of the parent node
(empty child node case)
Example:
Inserting a node D as the right child of B

Empty right child case:
- parent(B) -> rightThread = FALSE
- child(D) -> leftThread & rightThread = TRUE
- child -> leftChild = parent (1)
- child -> rightChild = parent -> rightChild (2)
- parent -> rightChild = child (3)

Example:
Inserting a node X as the right child of B

Non-empty right child case:

```c
void insertRight(threadedPointer parent, threadedPointer child)
{
    threadedPointer temp;
    (1) child->rightChild = parent->rightChild; child->rightThread = parent->rightThread;
    (2) child->leftChild = parent; child->leftThread = TRUE;
    (3) parent->rightChild = child; parent->rightThread = FALSE;
    if (!child->rightThread) /* non-empty child */
    (4) { temp = insucc(child); temp->leftChild = child; }
}
```
Heaps

- **Max tree** definition: a tree in which the key value in each node is no smaller than the key values in its children. A max heap is a complete binary tree that is also a max tree.

- **Mn tree** definition: a tree in which the key value in each node is no larger than the key values in its children. A min heap is a complete binary tree that is also a min tree.

- Application: priority queue
  - machine service: amount of time (min heap); amount of payment (max heap)
  - Factory: time tag

Abstract Data Type of Max Heap

structure MaxHeap
- objects: a complete binary tree of n > 0 elements organized so that the value in each node is at least as large as those in its children
- functions:
  - for all heap belong to MaxHeap, item belong to Element, n, max_size belong to integer
    - MaxHeap Create(max_size) ::= create an empty heap that can hold a maximum of max_size elements
    - Boolean HeapFull(heap, n) ::= if (n==max_size) return TRUE else return FALSE
    - MaxHeap Insert(heap, item, n) ::= if (!HeapFull(heap,n)) insert item into heap and return the resulting heap else return error
    - Boolean HeapEmpty(heap, n) ::= if (n>0) return FALSE else return TRUE
    - Element Delete(heap, n) ::= if (!HeapEmpty(heap,n)) return one instance of the largest element in the heap and remove it from the heap else return error
Heap Operations

- Creation of an empty heap
  - build a Heap $\mathcal{O}(n \log n)$
- Insertion of a new element into the heap
  - $\mathcal{O}(\log_2 n)$
- Deletion of the largest element from the max heap
  - $\mathcal{O}(\log_2 n)$

Application of Heap
- Priority Queues

<table>
<thead>
<tr>
<th>Representation</th>
<th>Insertion</th>
<th>Deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered array</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unordered linked list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$\mathcal{O}(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$\mathcal{O}(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Max heap</td>
<td>$\mathcal{O}(\log_2 n)$</td>
<td>$\mathcal{O}(\log_2 n)$</td>
</tr>
</tbody>
</table>

Insertion into a Max Heap

- Insert 5
- Insert 21
Algorithm of Insertion into a Max Heap

```c
void insert_max_heap(element item, int *n) // current size *n
{
    int i;
    if (HEAP_FULL(*n))
    {
        fprintf(stderr, "the heap is full."); exit(1);
    }
    i = ++(*n);
    while ((i!=1) && (item.key>heap[i/2].key))
    {
        heap[i] = heap[i/2];  i /= 2;
    }
    heap[i] = item;
}
```

- The height of n node heap = \(\lceil \log_2(n+1) \rceil\)
- Time complexity = \(O(\text{height}) = O(\log_2 n)\)

Deletion from a Max Heap

**Example: Delete the root of 20**
- Step 1: Remove the root
- Step 2: Replace the last element to the root
- Step 3: Heapify (Reestablish the heap)
Algorithm of Deletion from a Max Heap

```c
element delete_max_heap(int *n)
{
    int parent, child; element item, temp;
    if (HEAP_EMPTY(*n))
    { fprintf(stderr, "The heap is empty\n"); exit(1); }
    item = heap[1]; /* save value of the element with the highest key */
    /* use last element in heap to adjust heap */
    temp = heap[(*n)--]; parent = 1; child = 2;
    while (child <= *n) /* find the larger child of the current parent */
    { if ((child < *n) && (heap[child].key<heap[child+1].key))
        child++;
        if (temp.key >= heap[child].key) break;
        /* move to the next lower level */
        heap[parent] = heap[child]; parent= child; child *= 2;
    }
    heap[parent] = temp; return item;
}
```

Binary Search Trees

- Heap tree: search arbitrary element
  - $O(n)$ time
- Binary Search Trees (BST)
  - Searching $\Rightarrow O(h)$, $h$ is the height of BST
  - Insertion $\Rightarrow O(h)$
  - Deletion $\Rightarrow O(h)$
  - Can be done quickly by both key value and rank
Definition

A **binary search tree (BST)** is a binary tree, that may be empty or satisfies the following properties:

- Every element has a unique key.
- The keys in a nonempty left/right sub-tree must be smaller/larger than the key in the root of the sub-tree.
- The left and right sub-trees are also binary search trees.

**Example:** Three binary search trees

Two Algorithms of Searching a BST

```c
element * Recursive_search (tree_pointer root, int key)
{ /* return a pointer to the node that contains key. If there is no such
  node, return NULL */
  if (!root) return NULL;
  if (key == root->data) return root;
  if (key < root->data) return Recursive_search(root->leftChild,key);
  return Recursive_search(root->rightChild,key);
}

element * Iterative_search (tree_pointer tree, int key)
{ while (tree)
  { if (key == tree->data) return tree;
    if (key < tree->data) tree = tree->leftChild;
    else tree = tree->rightChild;
  }
  return NULL;
}
```

**O(h)**, $h$ is the height of BST.
Inserting into a BST

- Step 1: Check if the inserting key is different from those of existing elements
- Step 2: Run `insert_node` function

---

Algorithm of Inserting into a BST

```c
void insert_node(treePointer *node, int num) {
    treePointer ptr, temp = modifiedSearch(*node, num);
    if (temp || !(*node)) {
        MALLOC(ptr, sizeof(*ptr));
        ptr->data = num;
        ptr->leftChild = ptr->rightChild = NULL;
        if (*node)
            if (num < temp->data) temp->left_child = ptr;
            else temp->right_child = ptr;
        else *node = ptr;
    }
}
```
Deletion from a BST

- Delete a non-leaf node with two children
  - Replace the largest element in its left sub-tree
  - Or Replace the smallest element in its right sub-tree
  - Recursively to the leaf \( O(h) \)

![Tree Diagram](image)

Height of a BST

- The Height of the binary search tree is \( O(\log_2 n) \), on the average.
  - Worst case (skewed) \( O(h) = O(n) \)
- Balanced Search Trees
  - With a worst case height of \( O(\log_2 n) \)
  - AVL Trees, 2-3 Trees, and Red-Black Trees (Chapter 10)
Selection Trees

- Applications
  - Definition: A run is an ordered sequence
  - Merge $k$ ordered sequences into a single ordered sequence
  - Build a $k$-run selection tree
- Two kind of selection trees
  - Winner tree
  - Loser tree

Winner Tree

A winner tree is a complete binary tree that is represented using the sequential allocation scheme. Each node represents the smaller of its two children. The case $k = 8$.
Example: Operation in Winner Tree

After one record from run 4 (the key value of 15), the winner tree has been output and the tree restructured.

Time Complexity of Winner Tree

- Selection tree’s level
  - \( r \log_2 k + 1 \), \( k \) is number of runs
- Each time to restructure the tree
  - \( O(\log_2 k) \)
- Total time to merge \( n \) records
  - \( O(n \log_2 k) \)
Loser Tree

- The previous selection tree is called a **winner tree**
  - Each node records the winner of the two children

- **Loser Tree**
  - Leaf nodes represent the first record in each run
  - Each non-leaf node retains a pointer to the loser
  - Overall winner is stored in the additional node, node 0
  - Each newly inserted record is now compared with its parent (not its sibling) → loser stays, winner goes up without storing.
  - Slightly faster than winner trees

Example: Loser tree

Loser tree corresponding to winner tree

- Overall winner
- 9 = loser of winner-subtrees (9,6)
- 17 = loser of winner-subtrees (8,17)
**Forests**

- **Definition**: A **forest** is a set of $n \geq 0$ disjoint trees, $T_1, \ldots, T_n$.

![Diagram of a forest with trees $T_1$, $T_2$, and $T_3$]

---

**Transforming a Forest into a Binary Tree**

- Transforming a forest of trees, $T_1, \ldots, T_n$, into a binary tree $B(T_1, \ldots, T_n)$ and the algorithm is:
  1. If $n = 0$, then return empty.
  2. Has root equal to root($T_1$).
  3. Has left subtree equal to $B(T_{11}, T_{12}, \ldots, T_{1m})$.
  4. Has right subtree equal to $B(T_2, \ldots, T_n)$.

![Diagram showing transformation from forest to binary tree]
Forest Traversals

- **Preorder**
  1. If F is empty, then return
  2. Visit the root of the first tree of F
  3. Traverse the subtrees of the first tree in tree preorder
  4. Traverse the remaining trees of F in preorder

- **Inorder**
  1. If F is empty, then return
  2. Traverse the subtrees of the first tree in tree inorder
  3. Visit the root of the first tree
  4. Traverse the remaining trees of F in inorder

- **Postorder**
  1. If F is empty, then return
  2. Traverse the subtrees of the first tree of F in tree postorder
  3. Traverse the remaining trees of F in postorder
  4. Visit the root of the first tree of F

Representation of Disjoint Sets

- **Elements**: 0, 1, ..., n - 1.
- **Sets**: S_1, S_2, ..., S_m
  - pairwise disjoint: If S_i and S_j are two sets and i ≠ j, then there is no element that is in both S_i and S_j.

  S_1={0, 6, 7, 8}, S_2={1, 4, 9}, S_3={2, 3, 5}

- **Operations**
  - Disjoint Set Union: Ex: S_1 ∪ S_2={0,6,7,8,1,4,9}
  - Find (i): Find the set containing the element i. Ex: 3 ∈ S_3, 8 ∈ S_1

S_1 ∩ S_j = φ
Disjoint Set Union Operation

- Make one of trees a subtree of the other

Example: \( S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\} \)

Two possible representations

Implement of Data Structure

<table>
<thead>
<tr>
<th>Set name</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td></td>
</tr>
<tr>
<td>S_2</td>
<td></td>
</tr>
<tr>
<td>S_3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td>-1</td>
<td>4</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
**Simple Union and Find Operation**

```c
int simpleFind (int i)
{
    for (; parent[i] >= 0; i = parent[i]);
    return i;
}

void simpleUnion (int i, int j)
{
    parent[i] = j;
}
```

- **Performance**
  - `union(0,1), find(0)`
  - `union(1,2), find(0)`
  - `union(n-2,n-1), find(0)`

A degenerate tree can lead to a `union` operation being `O(n)` and a `find` operation being `O(n^2)`.

**Weighting Rule for Union**

- If `# of nodes in i < # of nodes in j`
  - Then `j` becomes the parent of `i`
  - Else `i` becomes the parent of `j`
**Modified Union Function**

- Prevent the tree from growing too high
  - To avoid the creation of degenerate trees
  - No node in T has level greater than $\lfloor \log_2 n \rfloor + 1$

```c
void weightUnion(int i, int j)
{
    int temp = parent[i]+parent[j];
    if (parent[i]>parent[j])
    {
        parent[i]=j; // make j the new root
        parent[j]=temp;
    } else
    {
        parent[j]=i; // make i the new root
        parent[i]=temp;
    }
}
```

**Trees Achieving Worst Case Bound**

$$\lfloor \log_2 n \rfloor + 1$$
Collapsing Rule for Modified Find

Definition: If $j$ is a node on the path from $i$ to its root then make $j$ a child of the root.

The new find function (see next slide):

- Roughly doubles the time for an individual find.
- Reduces the worse case time over a sequence of finds.

int collapsignFind (int i)
{
    int root, trail, lead;
    for (root=i; parent[root] >= 0; root=parent[root]) ;
    for (trail=i; trail!=root; trail=lead)
    {
        lead = parent[trail];
        parent[trail] = root;
    }
    return root;
}

Let $T(m, n)$ be the maximum time required to process an intermixed sequence of $m$ finds ($m \geq n$) and $n-1$ unions, we have:

$k_1m \alpha(m, n) \leq T(m, n) \leq k_2m \alpha(m, n)$; $k_1, k_2$ : some positive constants

$\alpha(m, n)$ is a very slowly growing function and is a functional inverse of Ackermann's function $A(p, q)$.

Function $A(p, q)$ is a very rapidly growing function.

Equivalence Classes

- Using union-find algorithms to processing the equivalence pairs.
- At most time: $O(m \alpha(2m, n))$
- Using less space
Counting Binary Trees

- Three disparate problems:
  - Having the same solution
  - Determine the number of distinct binary trees having $n$ nodes (problem 1)
  - Determine the number of distinct permutations of the numbers from 1 to $n$ obtainable by a stack (problem 2)
  - Determine the number of distinct ways of multiply $n + 1$ matrices (problem 3)

Distinct Binary Trees

- $n=1$: only one binary tree
- $n=2$: 2 distinct binary trees

- $n=3$: 5 distinct binary trees

- $n= ...$
Stack Permutations

- A binary tree traversals
  - Pre-order: A B C D E F G H I
  - In-order: B C A E D G H F I
  - Is this binary tree unique?

- Constructing a binary tree from its inorder and preorder sequences

Binary Trees Corresponding to Permutations

- For a given preorder permutation 1, 2, 3, what are the possible inorder permutations?
- Possible inorder permutation by a stack →
  - (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 2, 1)
  - (3, 1, 2) is impossible
- Each inorder permutation represents a distinct binary tree

Five permutations
Matrix Multiplication

- The product of n matrices: \( M_1 \cdot M_2 \cdot \ldots \cdot M_n \)
- Matrix multiplication is associative that can be performed in any order
  - \( n=3 \): 2 ways to perform
    \( (M_1 \cdot M_2) \cdot M_3 \) or \( M_1 \cdot (M_2 \cdot M_3) \)
  - \( n=4 \): 5 possibilities
    \( ((M_1 \cdot M_2) \cdot (M_3 \cdot M_4)) \) or \( (M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))) \)

- Let \( b_n \) be the number of different ways to compute the product of \( n \) matrices. We have
  \[
  b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1} \ , \ n \geq 1 \ , \text{and} \ b_0 = 1
  \]

Number of Distinct Binary Trees

- Approximation by solving the recurrence of the equation
  \[
  b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1} \ , \ n \geq 1 \ , \text{and} \ b_0 = 1
  \]

**Solution** (when \( x \to \infty \))

\[
\begin{align*}
\therefore B(x) &= \sum_{i=0}^{\infty} b_i \cdot x^i \\
&\Rightarrow xB^2(x) = B(x) - 1 \\
\therefore B(x) &= \frac{1}{2x} \left[ 1 - \sum_{n=0}^{\infty} \left( \frac{1}{2} \right)^n (-4x)^n \right] \\
&= \sum_{m=0}^{\infty} \left( \frac{1/2}{m+1} \right) (-1)^{2m+1} 2^{2m+1} x^{m+1}
\end{align*}
\]

Simplification: \( b_n = \frac{1}{n+1} \binom{2n}{n} \)

Approximation: \( b_n = O(4^n/n^{3/2}) \)

C-C Tsai  P.81

C-C Tsai  P.82