Chapter 5. TREES

1. INTRODUCTION
2. BINARY TREES
3. BINARY TREE TRAVERSALS
4. ADDITIONAL BINARY TREE OPERATIONS
5. THREADED BINARY TREES
6. HEAPS
7. BINARY SEARCH TREES
8. SELECTION TREES
9. FORESTS
10. SET REPRESENTATION
11. COUNTING BINARY TREES

1. INTRODUCTION

- Two Types of Genealogical Charts: Figure 5.1
- Definition of Tree:
  A finite set of one or more nodes s.t.:
  1. One Specially Designated Node called root
  2. Remaining Nodes: partitioned into $n \geq 0$ disjoint sets, $T_1, \ldots, T_n$, where each of $T_i$ is a tree.
  $T_1, \ldots, T_n$: Subtrees of root
A Sample Tree

Terminologies

- **Degree of a node (A: 3), Degree of a tree (3)**
- **Leaf or Terminal Node (K, L, F, G, M, I, J)**
- **Parent (E: B), Children (B: E & F), Siblings (E & F)**
- **Ancestor (M: H, D, A), Descendants (B: E, F, K, L)**
- **Level (Root: 1), Height or Depth (4)**

1.2 Representation of Trees

**List Representations**

- (A (B (E (K, L), F), C (G), D (H (M), I, J)))
- Using Linked List: Requires Varying # of Fields

<table>
<thead>
<tr>
<th>data</th>
<th>Link 1</th>
<th>Link 2</th>
<th>…</th>
<th>Link n</th>
</tr>
</thead>
</table>

**Left Child - Right Sibling Representation**

- Two Link Fields per Node

<table>
<thead>
<tr>
<th>data</th>
</tr>
</thead>
</table>

| Left child | Right child |
2. BINARY TREES

- The Abstract Data Type
- Properties of Binary Trees
- Binary Tree Representations
2.1 The Abstract Data Types

- **Chief Characteristics of Binary Tree**
  - Degree of Any Node Must Not Exceed 2
  - Distinguish Between Left Subtree and Right Subtree
  - Binary Tree May Have Zero Nodes

- **Definition of Binary Tree:**
  
  A finite set of nodes that is either
  1. Empty or
  2. A root with two disjoint binary trees, left/right subtree

![Figure 5.9: Skewed and complete binary trees](image-url)
**Structure 5.1: Abstract data type Binary_Tree**

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

**functions**:

- `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 data in the root node of bt.`
- `BinTree Rchild(bt) ::= if (IsEmpty(bt)) return error else return the right subtree of bt.`

**Figure 5.8: Two different binary trees**
2.2 Properties of Binary Trees

- **Lemma 5.1 [Maximum # of nodes]:**
  1. Maximum # of nodes on level \( i = 2^i - 1, i \geq 1 \)
  2. Maximum # of nodes in a binary tree of depth \( k \) = \[
  \sum_{i=1}^{k} 2^{i-1} = 2^k - 1
  \]

- **Lemma 5.2 [# of leaf nodes vs. # of nodes of degree 2]:**
  \( n_0 \): # of leaf nodes, \( n_2 \): # of nodes of degree 2
  \[ n_0 = n_2 + 1 \]
  (**Pf**) \( n = n_0 + n_1 + n_2 \) & \( n = B + 1 = n_1 + 2n_2 + 1 \)

- **Definition: Full binary tree of depth \( k \)**
  A binary tree of depth \( k \) having \( 2^k - 1 \) nodes, \( k \geq 0 \).

- **Definition: Complete binary tree of depth \( k \)**
  A binary tree of which nodes corresponds to the nodes numbered from 1 to \( n \) is in the full binary tree of depth \( k \).
2.3 Binary Tree Representations

- **Array Representation**
  Use 1-dimensional array by Lemma 5.3

- **Lemma 5.3**: Suppose a complete binary tree with \( n \) nodes
  is represented sequentially (depth = \( \lceil \log_2 n \rceil + 1 \)). Then for
  any node with index \( i \), \( 1 \leq i < n \), we have:
  1. \( \text{parent}(i) = \lfloor i/2 \rfloor \) if \( i \neq 1 \). If \( i = 1 \) (root), no parent.
  2. \( \text{lchild}(i) = 2i \) if \( 2i \leq n \). If \( 2i > n \), \( i \) has no left child.
  3. \( \text{rchild}(i) = 2i + 1 \) if \( 2i + 1 \leq n \). If \( 2i + 1 > n \), no right child.

- **Skewed Tree: waste space**

  ![Figure 5.11: Array representation of binary trees of Figure 5.9](image)
**Linked Representation**

```c
typedef struct node *tree_pointer;
typedef struct node {
    int data;
    tree_pointer left_child, right_child;
};
```

Figure 5.13: Linked representation for the binary trees of Figure 5.9
3. BINARY TREE TRAVERSAL

- **Problem Definition**
  - ✔ Visiting Each Node Exactly Once
  - ✔ Produce Linear Order in a Tree

```
void inorder(tree_pointer ptr)
{
    if (ptr) {
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        inorder(ptr->right_child);
    }
}
```

⇒ Output: A / B * C * D + E
**Preorder Traversal**

```c
void preorder(tree_pointer ptr)
{
    if (ptr) {
        // printf("%d", ptr->data);
        // preorder(ptr->left_child);
        // preorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

⇒ Output: + * * / A B C D E

**Postorder Traversal**

```c
void postorder(tree_pointer ptr)
{
    if (ptr) {
        postorder(ptr->left_child);
        postorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

⇒ Output: A B / C * D * E +

**Iterative Inorder Traversal:** Complexity = O(n)

**Level Order Traversal:** Breadth First Search

✔ + * E * D / C A B
void iter_inorder(tree_pointer node)
{
    int top = -1; /* initialize stack */
    tree_pointer stack[MAX_STACK_SIZE];
    for (; ;) {
        for (; node; node = node->left_child)
            add(&top, node);
        node = delete(&top);
        if (!node) break; /* empty stack */
        printf("%d", node->data);
        node = node->right_child;
    }
}

Program 5.4: Iterative Inorder Traversal

void level_order(tree_pointer ptr)
/* level order tree traversal */
{
    int front = rear = 0;
    tree_pointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    addq(front, &rear, ptr);
    for (; ;) {
        ptr = deleteq(&front, rear);
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->left_child) addq(front, &rear, ptr->left_child);
            if (ptr->right_child) addq(front, &rear, ptr->right_child);
        }
        else break;
    }
}

Program 5.5: Level order traversal of a binary tree
4. ADDITIONAL BINARY TREE OPERATION

- **Copying Binary Trees**
  - ✔ See Program 5.6: Modified Version of `postorder`
  - ✔ Design `swap_tree`: See Exercise 2 (Page 210)

- **Testing for Equality of Binary Trees**
  - ✔ See Program 5.7: Modified Version of `preorder`

- **The Satisfiability Problem**
  - ✔ Find Assignment of Values that Satisfy Expression
  - ✔ \((x_1 \& \neg x_2) \mid (\neg x_1 \& x_3) \mid \neg x_3\)
  - ✔ Generate \(2^n\) Combinations & Evaluate Each of them
  - ✔ See Program 5.8 and Program 5.9

```c
#define IS_NULL(ptr) ((ptr) == NULL)

tree_pointer copy(tree_pointer original)
/* this function returns a tree_pointer to an exact copy of the original tree */
{
    tree_pointer temp;
    if (original) {
        temp = (tree_pointer) malloc(sizeof(node));
        if (IS_NULL(temp)) {
            fprintf(stderr, “The memory is full/n”);
            exit(1);
        }
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        temp->data = original->data;
        return temp;
    } else {
        return NULL;
    }
}
```

Program 5.6: Copying a binary tree
int equal(tree_pointer first, tree_pointer second)
{
    /* function return 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))
    }

Program 5.7: Testing for equality of binary trees
**Node structure in C**

typedef enum {not, and, or, true, false} logical;
typedef struct node *tree_pointer;
typedef struct node {
    tree_pointer left_child, right_child;
    logical data;
    short int value;
};

for (all 2^n possible combinations) {
    generate the next combination;
    replace the variables by their values;
    evaluate root by traversing it in postorder;
    if (root->value) {
        printf(<combination>); return;
    }
}

**Program 5.8:** First version of satisfiability algorithm

```c
void post_order_eval(tree_pointer node)
{
    /* modified post order traversal to evaluate a propositional calculus tree */
    if (node) {
        post_order_eval(node->left_child);
        post_order_eval(node->right_child);
        switch(node->data) {
            case not:  node->value = !node->right_child->value; break;
            case and:  node->value = node->right_child->value &&
                        node->left_child->value;
                        break;
            case or:   node->value = node->right_child->value ||
                        node->left_child->value;
                        break;
            case true: node->value = TRUE; break;
            case false: node->value = FALSE;
        }
    }
}
```

**Program 5.9:** *post_order_eval* function
5. THREADED BINARY TREE

- **Basic Idea**
  - $n + 1$ null links out of $2n$ total links
  - Replace null links by pointers to other nodes
    \[ \Rightarrow \text{Threads} \]

- **Usage of Threads**
  - $\text{ptr} \rightarrow \text{left_child} = \emptyset$ : inorder predecessor of $\text{ptr}$
  - $\text{ptr} \rightarrow \text{right_child} = \emptyset$ : inorder successor of $\text{ptr}$

- **Node structure**
  ```c
  typedef struct thread_tree *threaded_pointer;
  typedef struct thread_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread;
  };
  ```

Figure 5.21: Threaded tree corresponding to Figure 5.9(b)
Head Node of Threaded Binary Tree

- Inorder predecessor of leftmost node
- Inorder successor of rightmost node

An Empty Threaded Binary Tree

<table>
<thead>
<tr>
<th>left_thread</th>
<th>left_child</th>
<th>data</th>
<th>right_child</th>
<th>right_thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>•</td>
<td>____</td>
<td>•</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

Figure 5.23: Memory representation of a threaded tree
Inorder Traversal of a Threaded Binary Tree

✔ If \( ptr->right_thread \) == TRUE
  ⇒ Inorder Successor of \( ptr = ptr->right_child \)
✔ Otherwise
  ⇒ Following left_child links from right_child of \( ptr \) until we reach a
    node with left_thread = TRUE
✔ Program 5.10 & Program 5.11: Complexity = \( O(n) \)
✔ Exercise 4/5: preorder/postorder traversal with threads

Inserting A Node Into A Threaded Binary Tree

✔ Insert new node as right child of parent
✔ Two cases: \( parent->right_thread \) is true or false
✔ See Figure 5.24 & Program 5.12
✔ Exercise 3: insert new node as left child of parent

threaded_pointer insucc(threaded_pointer tree)
{
  /* find the inorder successor of tree in a threaded binary tree */
  threaded_pointer temp;
  temp = tree->right_child;
  if (!tree->right_thread)
    while (!temp->left_thread)  temp = temp->left_child;
  return temp;
}

void tinorder (threaded_pointer tree)
{
  /* traverse the threaded binary tree inorder */
  threaded_pointer temp = tree;
  for (; ; ) {  
    temp = insucc(temp);
    if (temp = tree) break;
    printf("%3c", temp->data);
  }
}

Program 5.10: Finding the inorder successor of a node
Program 5.11: Inorder traversal of a threaded binary tree
Figure 5.24 Insertion of child as a right child of parent in a threaded binary tree
void insert_right(threaded_pointer parent, threaded_pointer child) {
    /* insert child as the right child of parent in a threaded binary tree */
    threaded_pointer temp;
    child->right_child = parent->right_child;
    child->right_thread = parent->right_thread;
    child->left_child = parent;
    child->left_thread = TRUE;
    parent->right_child = child;
    parent->right_thread = FALSE;
    if (!child->right_thread) {
        temp = insucc(child);
        temp->left_child = child;
    }
}

Program 5.12: Right insertion in a threaded binary tree

---

6. HEAPS

- The Heap Abstract Data Type
- Priority Queues
- Insertion into a Max Heap
- Deletion from a Max Heap
6.1 The Heap Abstract Data Type

- **Definition: max tree & max heap**
  - max tree: key value of a node \( \geq \) key values of children
  - max heap: complete binary tree & max tree

- **Definition: min tree & min heap**
  - min tree: key value of a node \( \leq \) key values of children
  - min heap: complete binary tree & min tree

6.2 Priority Queues

- **What is a priority queue?**
  - Deletes an element with highest (or lowest) priority

- **Priority Queue Representations**

<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>( O(n) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>( O(n) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Max heap</td>
<td>( O(\log_2 n) )</td>
<td>( O(\log_2 n) )</td>
</tr>
</tbody>
</table>
Figure 5.25: Sample max heaps

Figure 5.26: Sample min heaps

structure MaxHeap is
  objects: a complete binary tree of \( n \geq 0 \) elements organized so that the value in each node is at least as large as those in its children
  functions:
    for all heap \( \in \text{MaxHeap}, \text{item} \in \text{Element}, n, \text{max}_\text{size} \in \text{integer} \)
    \[
    \text{MaxHeap Create}(\text{max}_\text{size}) ::= \text{create an empty heap that can hold a maximum of } \text{max}_\text{size} \text{ elements.}
    \]
    \[
    \text{Boolean HeapFull}(\text{heap}, n) ::= \begin{cases} \text{if } (n == \text{max}_\text{size}) & \text{return TRUE} \\ \text{else} & \text{return FALSE} \end{cases}
    \]
    \[
    \text{MaxHeap Insert}(\text{heap}, \text{item}, n) ::= \begin{cases} \text{if } (!\text{HeapFull}(\text{heap}, n)) & \text{insert } \text{item} \text{ into } \text{heap} \text{ and return the resulting heap } \text{heap} \\ \text{else} & \text{return error.} \end{cases}
    \]
    \[
    \text{Boolean HeapEmpty}(\text{heap}, n) ::= \begin{cases} \text{if } (n > 0) & \text{return FALSE} \\ \text{else} & \text{return TRUE} \end{cases}
    \]
    \[
    \text{Element Delete}(\text{heap}, n) ::= \begin{cases} \text{if } (!\text{HeapEmpty}(\text{heap}, n)) & \text{return one instance of the largest element in the heap and remove it from the heap } \text{heap} \\ \text{else} & \text{return error.} \end{cases}
    \]

Structure 5.2: Abstract data type MaxHeap
6.3 Insertion Into A Max Heap

- **Use Array Representation**
  - Insert new node to last position
  - Compare & Replace it until root

- **C Declarations**
  ```c
  #define MAX_ELEMENTS 200 /*maximum heap size+1 */
  #define HEAP_FULL(n) (n == MAX_ELEMENTS-1)
  #define HEAP_EMPTY(n) (!n)
  typedef struct {
    int key;
    /* other fields */
  } element;
  
  element heap[MAX_ELEMENTS];
  int n = 0;
  ``

- **See Figure 5.28 & Program 5.13: Complexity = O(log₂n)**

---

**Figure 5.28: Insertion into a max heap**
Void insert_max_heap(element item, int *n) {
    /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(1);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}

Program 5.13: Insertion into a max heap

6.4 Deletion From A Max Heap

- Basic Idea
  - Delete root node & Move last node to root
  - Reestablish the heap by moving down & comparing

- See Figure 5.29 & Program 5.14:
  - Complexity = O(log_2 n)
Figure 5.29: Deletion from a max heap

Program 5.14: Deletion from a max heap
7. BINARY SEARCH TREES

- **Problems of Heap**
  - ✔ Deletion of arbitrary element: $O(n)$
  - ✔ Searching for an arbitrary element: $O(n)$

- **Definition: binary search tree**
  - ✔ Every element has a unique key
  - ✔ Keys in a nonempty left subtree $<$ Key of root
  - ✔ Keys in a nonempty right subtree $>$ Key of root
  - ✔ Left & right subtrees are also binary search trees

---

![Binary trees](image_url)

Figure 5.30: **Binary trees**
7.1 Searching a Binary Search Tree

Basic Idea
✔ If (key == root->key) return(root)
✔ If (key < root->key) search(root->left_child)
   Otherwise, search(root->right_child)
✔ Program 5.15 (recursive) & Program 5.16 (iterative)
✔ Complexity: O(height of tree)

---

```c
void 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 search (root->left_child, key);
    return search (root->right_child, key);
}
```

**Program 5.15:** Recursive search of a binary search tree
Program 5.16: Iterative search of a binary search tree

```c
tree_pointer search2 (tree_pointer tree, int key) {
    /* return a pointer to the node that contains key. 
       If there is no such node, return NULL. */
    while (tree) {
        if (key == tree->data) return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
    }
    return NULL;
}
```

7.2 Inserting Into a Binary Search Tree

- **Basic Idea**
  - Search tree to verify key uniqueness
  - Insert element where the search is terminated
  - modified_search(root, num)
    - If num is present, return(NULL)
    - Otherwise, return pointer of last node during search
  - Program 5.17: Complexity = O(height of tree)
Figure 5.31: Inserting into a binary search tree

(a) Insert 80

(b) Insert 35

Program 5.17: Inserting an element into a binary search tree
7.3 Deletion From a Binary Search Tree

- **Basic Idea**
  - Deletion of leaf node: `parent->left_child = NULL`
    - (Ex) Delete 35 from Figure 5.31(b)
  - Deletion of non-leaf node that has only a single child
    - Place child in the place of erased node
    - (Ex) Delete 40 from Figure 5.31(a)
  - Deletion of non-leaf node with two children
    - Replace node with largest element in left subtree
      or with smallest element in right subtree
    - See Figure 5.33

---

**Figure 5.33**: Deletion of a node with two children
7.4 Height of a Binary Search Tree

- Worst case: $O(n)$, Average case: $O(\log_2 n)$
- Balanced Binary Search Tree: AVL Tree, B Tree

9. FORESTS

- **Definition**: *forest*
  set of $n \geq 0$ disjoint trees

- **Definition**: *binary tree representation of forest* $T_1, ..., T_n$
  (1) is empty if $n = 0$
  (2) root = root($T_1$), left subtree = subtree of $T_1$
      right subtree = B($T_2, ..., T_n$)
Preorder Traversal of Forest $F$
1. If $F$ is empty, then return
2. Visit root of first tree of $F$
3. Traverse subtrees of first tree in preorder
4. Traverse remaining trees of $F$ in preorder

Inorder Traversal of Forest $F$
1. If $F$ is empty, then return
2. Traverse subtrees of first tree in inorder
3. Visit root of first tree of $F$
4. Traverse remaining trees of $F$ in inorder
10. SET REPRESENTATION

- **Assumptions**
  - ✔ The elements of the sets are the number 0, 1, 2, ..., \( n-1 \).
  - ✔ The sets being represented are pairwise disjoint, that is, if \( S_i \) and \( S_j \) are two sets and \( i \neq j \), then there is no element that is in both \( S_i \) and \( S_j \).

- **Minimal operations on these sets:**
  1. *Disjoint set union*
     \( S_i \) and \( S_j \) two disjoint sets
     \[ S_i \cup S_j = \{ x \mid x \in S_i \text{ or } x \in S_j \} \]
  2. *Find\((i)\)*
     Find the set containing the element, \( i \).

- **Representation of sets**
  ✔ \( S_1 = \{ 0, 6, 7, 8 \} \), \( S_2 = \{ 1, 4, 9 \} \), \( S_3 = \{ 2, 3, 5 \} \)
union \( (S_1, S_2) \)

\[ S_1 \cup S_2 \]

\[ S_2 \cup S_1 \]

Data Representation of \( S_1, S_2 \) and \( S_3 \)
Array representation of $S_1$, $S_2$, and $S_3$

<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>

```
int find1(int i)
{
    for (; parent[i] >= 0; i = parent[i]) ;
    return i;
}

void union1(int i, int j)
{
    parent[i] = j;
}
```

Degenerate tree

- union(0, 1), union(1, 2), ..., union($n - 2$, $n - 1$)
**Definition: Weighting rule for union(i, j)**

- If the number of nodes in tree \(i\) is less than the number in tree \(j\), then make \(j\) the parent of \(i\); otherwise make \(i\) the parent of \(j\).

---

**void union2 (int n, int j)**

```c
{" union the sets with roots i and j, i != j, using the weighting rule.
parent[i] = -count[i] and parent[j] = -count[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;
}
"
)
```

Program 5.19 : Union function
Lemma 5.4 [Complexity of union2]  

✔ T: tree with n nodes created as a result of union2  
⇒ No node in T has level greater than ⌊log_2 n⌋+ 1

Definition [Collapsing Rule]  

✔ If j is a node on the path from i to its root, then make j a child of the root.

```c
int find2(int i) {
    /* find the root of the tree containing element i. Use the collapsing rule to collapse all nodes from i to root */
    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;
}
```

Program 5.20: Find function
(Ex) 8 find(7)
- without collapsing: 24 moves
- With collapsing: 12 moves

- Equivalent Classes
  - Two finds & at most one union
  - Efficient Space Use: O(m + n) vs. O(n)
  - See Figure 5.46
11. COUNTING BINARY TREES

- Constructing Binary Tree
  - Preorder: \(ABCDEF\) \(\leftrightarrow\) Postorder?
  - Inorder: \(BCAE\)

- The Followings are Equal
  - # of distinct binary trees having \(n\) nodes
  - # of distinct stack permutations for \(n\) data
  - # of distinct ways of multiplying \(n+1\) matrices