data structures a pseudocode approach with c

Data Structures: A Pseudocode Approach with C

Understanding data structures is fundamental for any aspiring programmer, and exploring them through a pseudocode approach with C offers a robust and accessible learning path. This article delves into the core concepts of various data structures, illustrating their logic and implementation using pseudocode, and then connects these abstract ideas to their practical realization in the C programming language. We will navigate through the building blocks of efficient data management, covering linear structures like arrays and linked lists, hierarchical structures such as trees, and more complex graph representations. By understanding the underlying principles via pseudocode and then seeing how these translate into C code, you'll gain a deeper appreciation for how to organize and manipulate data effectively in your software development endeavors.

Understanding the Importance of Data Structures

Data structures are the bedrock of efficient software development. They are essentially systematic ways of organizing, managing, and storing data in a computer so that it can be accessed and manipulated effectively. The choice of data structure significantly impacts the performance of algorithms, dictating how quickly operations like searching, insertion, and deletion can be performed. In the realm of computer science, mastering different data structures is not just about memorizing definitions; it's about understanding the trade-offs and choosing the most appropriate tool for a given problem. A solid grasp of data structures a pseudocode approach with C provides a versatile foundation, enabling developers to build scalable and optimized applications.

When considering software applications, from simple data retrieval to complex simulations, the efficiency with which data can be processed is paramount. Poorly chosen data structures can lead to sluggish performance, increased memory consumption, and a generally suboptimal user experience. Conversely, a well-chosen data structure can dramatically improve speed and resource utilization. This article aims to demystify these essential concepts, offering a clear path to understanding through the intuitive lens of pseudocode, followed by practical application in the widely used C programming language.

Linear Data Structures: The Foundation

Linear data structures are characterized by the sequential arrangement of their elements, where each element is connected to its preceding and succeeding element. This sequential nature makes them intuitive to understand and implement for many common programming tasks. We will explore

arrays and linked lists, two of the most fundamental linear data structures, using pseudocode to illustrate their behavior.

Arrays: Sequential Storage

An array is a collection of elements of the same data type, stored in contiguous memory locations. Each element is identified by an index, typically starting from zero. Arrays are excellent for random access, meaning you can directly access any element if you know its index. However, their size is usually fixed upon declaration, which can be a limitation.

Pseudocode for Array Declaration and Access:

- `DECLARE an_array AS ARRAY OF INTEGER with SIZE N`
- `SET an_array[0] = 10`
- `SET an_array[5] = 25`
- `PRINT an_array[0] // Output: 10`
- `PRINT an_array[5] // Output: 25`

In C, an array declaration would look like: `int an_array[10];`, and accessing an element would be `an_array[0] = 10;`. The fixed-size nature of C arrays means that if you need more space than initially allocated, you might have to reallocate memory, which can be an expensive operation. However, for scenarios where the size is known beforehand, arrays offer excellent performance for element access.

Linked Lists: Dynamic Sequences

Linked lists offer a more flexible alternative to arrays, especially when dealing with dynamic collections of data where insertions and deletions are frequent. Instead of contiguous memory, each element, called a node, contains the data itself and a pointer to the next node in the sequence. This pointer-based approach allows the list to grow and shrink dynamically.

Singly Linked Lists

In a singly linked list, each node points only to the next node. This makes traversal straightforward in one direction. Operations like insertion at the beginning or end are relatively efficient. Insertion in the middle requires locating the preceding node, and deletion involves adjusting the pointer of the previous node.

Pseudocode for Singly Linked List Operations:

- `STRUCTURE Node:`
    - `DATA value`
    - `POINTER next_node`
- `DECLARE head_node AS POINTER to Node, initially NULL`
- `PROCEDURE insert_at_beginning(data_value):`
    - `CREATE new_node AS Node`
    - `SET new_node.value = data_value`
    - `SET new_node.next_node = head_node`
    - `SET head_node = new_node`
- `PROCEDURE display_list():`
    - `DECLARE current_node AS POINTER to Node, initially head_node`
    - `WHILE current_node IS NOT NULL:`
        - `PRINT current_node.value`
        - `SET current_node = current_node.next_node`
    - `END WHILE`

In C, a singly linked list would be implemented using a struct: `struct Node { int data; struct Node next; };`. Memory for each node is typically allocated dynamically using `malloc()`. The `head_node` would be a pointer to the first node. This dynamic allocation is key to the flexibility of linked lists.

Doubly Linked Lists

Doubly linked lists enhance singly linked lists by providing each node with two pointers: one to the

next node and another to the previous node. This bidirectional linkage allows for more efficient traversal in both forward and backward directions, and simplifies operations like deletion, as you have direct access to the previous node without needing to traverse from the head.

Pseudocode for Doubly Linked List Node:

- `STRUCTURE DoublyNode:`
    - `DATA value`
    - `POINTER next_node`
    - `POINTER previous_node`

Implementing this in C involves a similar struct definition: `struct DoublyNode { int data; struct DoublyNode next; struct DoublyNode prev; };`. The extra pointer adds a slight overhead in terms of memory per node, but the benefits in terms of traversal and manipulation flexibility can be significant, especially in applications requiring frequent backward navigation or easy deletion of arbitrary nodes.

Circular Linked Lists

Circular linked lists are a variation where the last node's pointer points back to the first node (the head), forming a circle. This structure is useful in scenarios where a continuous loop is desired, such as in round-robin scheduling algorithms. Traversal can continue indefinitely, and any node can be a starting point for traversing the entire list.

Pseudocode for Circular Linked List Traversal:

- `PROCEDURE traverse_circular_list(start_node):`
    - `DECLARE current_node AS POINTER to Node, initially start_node`
    - `REPEAT:`
        - `PRINT current_node.value`
        - `SET current_node = current_node.next_node`
    - `UNTIL current_node IS start_node`

In C, this would involve ensuring the `next` pointer of the last node points to the first node. The termination condition in loops or traversal functions becomes critical to avoid infinite loops. The ability to start traversal from any node is a unique characteristic of circular linked lists.

Non-Linear Data Structures: Beyond Sequences

Non-linear data structures, unlike their linear counterparts, do not arrange elements in a sequential manner. Instead, elements can be connected to multiple other elements, allowing for more complex relationships and hierarchical organization. This section will explore stacks, queues, trees, and graphs.

Stacks: Last-In, First-Out (LIFO)

A stack is a data structure that follows the Last-In, First-Out (LIFO) principle. Imagine a stack of plates; you can only add a new plate to the top, and you can only remove the topmost plate. The primary operations are PUSH (adding an element to the top) and POP (removing the top element).

Pseudocode for Stack Operations:

- `DECLARE stack_array AS ARRAY OF ELEMENT with MAX_SIZE`
- `DECLARE top_index AS INTEGER, initially -1`
- `PROCEDURE push(element):`
    - `IF top_index < MAX_SIZE - 1:`
        - `INCREMENT top_index`
        - `SET stack_array[top_index] = element`
    - `ELSE:`
        - `PRINT "Stack Overflow"`
    - `END IF`
- `PROCEDURE pop():`

    - IF top_index >= 0:
        - DECLARE popped_element = stack_array[top_index]
        - DECREMENT top_index
        - RETURN popped_element
    - ELSE:
        - PRINT "Stack Underflow"
        - RETURN NULL (or an error indicator)
    - END IF

In C, a stack can be implemented using an array or a linked list. The array implementation is straightforward with a `top` index. The linked list implementation uses nodes, with `top` pointing to the most recently added node. Stacks are crucial for function call management, expression evaluation, and backtracking algorithms.

Queues: First-In, First-Out (FIFO)

A queue operates on the First-In, First-Out (FIFO) principle, similar to a waiting line. The element that has been in the queue the longest is the first to be removed. The primary operations are ENQUEUE (adding an element to the rear) and DEQUEUE (removing an element from the front).

Pseudocode for Queue Operations:

- DECLARE queue_array AS ARRAY OF ELEMENT with MAX_SIZE
- DECLARE front_index AS INTEGER, initially 0
- DECLARE rear_index AS INTEGER, initially -1
- PROCEDURE enqueue(element):
    - IF rear_index < MAX_SIZE - 1:

        - INCREMENT rear_index
        - SET queue_array[rear_index] = element
        - IF front_index == -1: // If it's the first element
            - SET front_index = 0
        - END IF
    - ELSE:
        - PRINT "Queue Overflow"
    - END IF
- PROCEDURE dequeue():
    - IF front_index != -1 AND front_index <= rear_index:
        - DECLARE dequeued_element = queue_array[front_index]
        - IF front_index == rear_index: // If it's the last element
            - SET front_index = -1
            - SET rear_index = -1
        - ELSE:
            - INCREMENT front_index
        - END IF
        - RETURN dequeued_element
    - ELSE:

        - PRINT "Queue Underflow"
        - RETURN NULL (or an error indicator)
    - END IF

In C, queues can be implemented using arrays (circular arrays are common to avoid wasted space) or linked lists. The linked list implementation typically uses pointers to the front and rear nodes. Queues are used in operating systems for task scheduling, managing requests in web servers, and breadth-first searches.

Trees: Hierarchical Organization

Trees are hierarchical data structures where elements are organized in a parent-child relationship. Each node can have zero or more child nodes. The topmost node is called the root, and nodes without children are called leaves. Trees are ideal for representing hierarchical data, such as file systems or organizational charts.

Binary Trees

A binary tree is a special type of tree where each node has at most two children, referred to as the left child and the right child. This structure is fundamental to many other more specialized tree types.

Pseudocode for Binary Tree Node:

- STRUCTURE TreeNode:
    - DATA value
    - POINTER left_child
    - POINTER right_child

In C, this translates to: `struct TreeNode { int data; struct TreeNode left; struct TreeNode right; };`. Operations like insertion, deletion, and searching in binary trees are often recursive or iterative, involving traversing the tree based on specific criteria.

Binary Search Trees (BST)

A Binary Search Tree (BST) is a binary tree with an additional property: for any given node, all values in its left subtree are less than the node's value, and all values in its right subtree are greater than the node's value. This property enables efficient searching, insertion, and deletion operations, typically with an average time complexity of O(log n).

Pseudocode for BST Insertion:

- `PROCEDURE insert_bst(root, new_value):`
    - `IF root IS NULL:`
        - `CREATE new_node with value new_value`
        - `RETURN new_node`
    - `ELSE IF new_value < root.value:`
        - `SET root.left_child = insert_bst(root.left_child, new_value)`
    - `ELSE IF new_value > root.value:`
        - `SET root.right_child = insert_bst(root.right_child, new_value)`
    - `END IF`
    - `RETURN root`

In C, implementing a BST involves carefully managing pointers and recursion to maintain the BST property. Balancing the tree, as done in AVL trees or Red-Black trees, is crucial to prevent worst-case scenarios where the tree degenerates into a linked list, degrading performance.

AVL Trees

AVL trees are self-balancing binary search trees. They maintain a balanced structure by performing rotations whenever an insertion or deletion causes an imbalance. This ensures that the height difference between the left and right subtrees of any node is at most one, guaranteeing O(log n) time complexity for most operations.

Pseudocode Concept for AVL Tree Balancing:

- `PROCEDURE balance(node):`
    - `CALCULATE balance_factor = height(node.left_child) - height(node.right_child)`
    - `IF balance_factor > 1: // Left heavy`
        - `IF balance_factor_of(node.left_child) < 0: // Left-Right Case`
            - `PERFORM left_rotation(node.left_child)`
        - `PERFORM right_rotation(node)`
    - `ELSE IF balance_factor < -1: // Right heavy`
        - `IF balance_factor_of(node.right_child) > 0: // Right-Left Case`
            - `PERFORM right_rotation(node.right_child)`
        - `PERFORM left_rotation(node)`
    - `END IF`
    - `RETURN node`

Implementing AVL tree balancing in C involves intricate pointer manipulation and recursive calls to maintain the tree's height balance after insertions and deletions. The complexity arises from identifying imbalance cases and executing the appropriate rotations.

Heaps

A heap is a specialized tree-based data structure that satisfies the heap property. In a min-heap, the parent node is always less than or equal to its children, while in a max-heap, the parent node is always greater than or equal to its children. Heaps are commonly used for priority queues and heap sort algorithms.

Pseudocode for Min-Heap Property:

- PROCEDURE heapify_down(heap, index, heap_size):
    - DECLARE smallest = index
    - DECLARE left_child_index = 2 index + 1
    - DECLARE right_child_index = 2 index + 2
    - IF left_child_index < heap_size AND heap[left_child_index] < heap[smallest]:
        - SET smallest = left_child_index
    - IF right_child_index < heap_size AND heap[right_child_index] < heap[smallest]:
        - SET smallest = right_child_index
    - IF smallest != index:
        - SWAP heap[index] and heap[smallest]
        - heapify_down(heap, smallest, heap_size)
    - END IF

Heaps are often implemented using arrays for efficiency. The parent-child relationships can be calculated directly from array indices. In C, you would use an array and functions to maintain the heap property during insertions and deletions. The `heapify` operation is central to heap management.

Graphs: Interconnected Relationships

Graphs are non-linear data structures that consist of a set of vertices (or nodes) and a set of edges connecting pairs of vertices. They are used to model relationships between objects, such as social networks, road maps, or dependencies between tasks. Graphs can be directed or undirected, and can contain weights on their edges.

Representing Graphs

Two common ways to represent graphs are adjacency matrices and adjacency lists.

- **Adjacency Matrix:** A 2D array where `matrix[i][j] = 1` if there is an edge from vertex `i` to vertex `j`, and `0` otherwise. For weighted graphs, it stores the weight. This is efficient for dense graphs but can consume a lot of memory for sparse graphs.
- **Adjacency List:** An array of linked lists, where each index `i` corresponds to a vertex, and the linked list at that index contains all vertices adjacent to vertex `i`. This is more memory-efficient for sparse graphs.

Pseudocode for Adjacency List Representation:

- `DECLARE graph AS ARRAY OF LINKED LISTS, size V (number of vertices)`
- `PROCEDURE add_edge(u, v): // For undirected graph`
    - `ADD v to the linked list at graph[u]`
    - `ADD u to the linked list at graph[v]`

In C, an adjacency list can be implemented using an array of pointers to linked list nodes, where each node stores the adjacent vertex. For example: `struct AdjListNode { int vertex; struct AdjListNode next; }; struct Graph { int V; struct AdjListNode adjLists; };`. Adjacency matrices are simpler to implement as 2D arrays: `int adjMatrix[V][V];`.

Graph Traversal Algorithms

Two fundamental graph traversal algorithms are Breadth-First Search (BFS) and Depth-First Search (DFS).

- **BFS:** Explores a graph layer by layer. It uses a queue to keep track of vertices to visit. Starting from a source vertex, it visits all its neighbors, then all their unvisited neighbors, and so on.
- **DFS:** Explores as far as possible along each branch before backtracking. It typically uses a stack (or recursion) to keep track of vertices to visit.

Pseudocode for BFS:

- PROCEDURE bfs(start_vertex):
    - CREATE a queue Q
    - MARK start_vertex as visited
    - ENQUEUE start_vertex into Q
    - WHILE Q is not empty:
        - current_vertex = DEQUEUE from Q
        - PROCESS current_vertex
        - FOR EACH neighbor of current_vertex:
            - IF neighbor is not visited:
                - MARK neighbor as visited
                - ENQUEUE neighbor into Q
            - END IF
        - END FOR
    - END WHILE

Implementing BFS and DFS in C involves managing visited arrays and using appropriate data structures (queues for BFS, recursion/stacks for DFS) to track the traversal path. These algorithms are foundational for solving many graph problems, including finding shortest paths, cycle detection, and topological sorting.

Choosing the Right Data Structure

Selecting the appropriate data structure is a critical design decision that directly impacts the efficiency and scalability of your programs. The ideal choice depends on the specific operations you need to perform and the nature of the data itself. For instance, if you require rapid access to elements by index and your dataset size is predictable, an array is often suitable. However, if your

data collection frequently grows or shrinks, or if frequent insertions and deletions are necessary, linked lists might be a better fit.

Consider the following when making your selection:

- **Frequency of Operations:** How often will you be searching, inserting, deleting, or traversing?
- **Data Size and Growth:** Is the dataset fixed, or will it change dynamically?
- **Memory Constraints:** Some structures have higher memory overhead than others.
- **Order of Elements:** Does the order matter, or is it a collection of unrelated items?
- **Specific Problem Requirements:** Certain algorithms or problems lend themselves naturally to particular data structures (e.g., stacks for recursion, queues for breadth-first traversal).

For example, if you're building a system that needs to quickly find the minimum or maximum element among a changing set, a heap would be a strong candidate. If you are implementing a spell checker or a symbol table, a hash table or a trie might offer superior performance for lookups.

Pseudocode to C: Bridging the Gap

The transition from pseudocode to C is a crucial step in solidifying your understanding of data structures. Pseudocode provides a high-level, language-agnostic description of an algorithm or data structure's logic, making it easier to grasp the core concepts without getting bogged down in syntax. C, being a low-level but powerful language, allows for direct manipulation of memory and provides the necessary tools to implement these concepts efficiently.

When converting pseudocode to C:

- **Abstract Concepts to Concrete Syntax:** Translate terms like "DECLARE," "SET," "PROCEDURE," and "STRUCTURE" into C keywords and constructs like variable declarations, assignments, function definitions, and `struct` definitions.
- **Pointers and Memory Management:** C's powerful pointer system is essential for implementing linked lists, trees, and graphs. You'll use `malloc()` and `free()` for dynamic memory allocation, which corresponds to the `CREATE` and `DELETE` operations in pseudocode.
- **Control Flow:** Pseudocode loops (`WHILE`, `FOR`, `REPEAT-UNTIL`) and conditional statements (`IF-THEN-ELSE`) directly map to C's `while`, `for`, `do-while`, `if`, and `switch` statements.
- **Data Types:** Ensure that the data types used in pseudocode (e.g., INTEGER, ELEMENT) are

mapped to appropriate C types (e.g., `int`, `char`, `float`, or custom `struct`s).

This structured approach, moving from abstract pseudocode to concrete C implementation, builds a strong foundation for tackling complex programming challenges. The C language's efficiency and control make it an excellent choice for understanding how data structures are managed at a more fundamental level.

Conclusion

The journey through data structures a pseudocode approach with C illuminates the intricate ways data can be organized and manipulated for optimal program performance. From the fundamental sequential nature of arrays and linked lists to the hierarchical relationships modeled by trees and the complex interconnections of graphs, each structure serves a distinct purpose. By visualizing these concepts through pseudocode and understanding their tangible implementation in C, developers gain a powerful toolkit for designing efficient, scalable, and robust software solutions. Mastering these core data structures is an ongoing process, but the foundation laid here provides a clear pathway to effectively managing and processing information in any programming context.

Frequently Asked Questions

What is the advantage of using pseudocode when explaining data structures?

Pseudocode offers a language-agnostic, high-level description of algorithms and data structure operations. It allows developers to focus on the logic and steps involved without getting bogged down in specific C syntax, making it easier to understand, communicate, and translate to various programming languages.

How can pseudocode effectively represent a Linked List in C?

Pseudocode for a Linked List typically outlines the structure of a node (containing data and a pointer to the next node) and functions for operations like insertion, deletion, and traversal. For example, `INSERT(list, value)` might be described as: `newNode = CREATE_NODE(value); IF list is empty THEN list.head = newNode; ELSE find last node; lastNode.next = newNode; END IF`.

Explain the pseudocode approach for implementing a Stack in C.

A Stack's pseudocode operations are straightforward. For `PUSH(stack, element)`: `IF stack is not full THEN add element to top; increment stack size; ELSE report overflow; END IF`. For `POP(stack)`: `IF stack is not empty THEN remove and return element from top; decrement stack size; ELSE report underflow; END IF`.

What pseudocode would describe a Queue's enqueue and dequeue operations in C?

Enqueueing adds an element to the rear, and dequeuing removes from the front. Pseudocode for `ENQUEUE(queue, element)`: `IF queue is not full THEN add element to rear; update rear pointer; increment queue size; ELSE report overflow; END IF`. For `DEQUEUE(queue)`: `IF queue is not empty THEN remove and return element from front; update front pointer; decrement queue size; ELSE report underflow; END IF`.

How does pseudocode simplify the explanation of tree traversals (e.g., Inorder) in C?

Pseudocode for Inorder traversal of a Binary Tree would be: `INORDER(node): IF node is not NULL THEN INORDER(node.left); PRINT node.data; INORDER(node.right); END IF`. This clearly shows the recursive nature of visiting the left subtree, then the current node, then the right subtree, independent of C's pointer syntax.

Can you show pseudocode for finding an element in a Binary Search Tree (BST) in C?

Certainly. Pseudocode for searching a BST: `SEARCH(node, key): IF node is NULL THEN return NOT_FOUND; IF key equals node.data THEN return FOUND; IF key is less than node.data THEN return SEARCH(node.left, key); ELSE return SEARCH(node.right, key); END IF`.

What is the pseudocode representation for implementing a Hash Table's insertion with collision handling (e.g., separate chaining) in C?

For separate chaining, pseudocode for `INSERT(hashTable, key, value)` might be: `index = HASH(key); IF hashTable[index] is empty THEN create a new linked list with (key, value); ELSE append (key, value) to the linked list at hashTable[index]; END IF`. This highlights the concept of storing multiple entries at a single hash index.

How can pseudocode effectively illustrate the concept of sorting algorithms like Bubble Sort when implemented in C?

Bubble Sort pseudocode could look like: `FOR i from 0 to n-2 DO FOR j from 0 to n-2-i DO IF array[j] > array[j+1] THEN SWAP(array[j], array[j+1]); END IF END FOR END FOR`. This visualizes the pairwise comparisons and swaps central to the algorithm, making its efficiency and logic clear.

Additional Resources

Here are 9 book titles, starting with *and related to data structures and pseudocode with a C flavor, along with their descriptions:*

1. Implementing Algorithms with C and Pseudocode

This book serves as a foundational guide to understanding and implementing common data structures. It bridges the gap between abstract algorithmic concepts and concrete C-like pseudocode, making complex ideas more accessible. Readers will learn to think algorithmically and express solutions clearly, preparing them for actual C programming.

2. Data Structures: A Pseudocode Journey with C in Mind
Embark on a conceptual journey through the landscape of data structures, from simple arrays to sophisticated trees and graphs. This text emphasizes a pseudocode approach, allowing for a deep understanding of the logic before diving into specific language syntax. The C programming language is used as a reference point for implementation discussions, providing practical context.

3. Algorithmic Thinking with Pseudocode and C Foundations
Develop strong algorithmic thinking skills by exploring fundamental data structures and their operations. The book utilizes clear, consistent pseudocode as its primary representation, with detailed explanations of how these structures would translate into C code. It focuses on the "why" and "how" of data organization, building a solid base for further programming studies.

4. Understanding Data Structures Through C-Inspired Pseudocode
This title offers an intuitive exploration of essential data structures, relying heavily on C-inspired pseudocode for clarity. Each concept is broken down into logical steps, illustrating the flow of operations and memory management considerations relevant to C. The goal is to foster a deep conceptual grasp of how data is organized and manipulated efficiently.

5. The Art of Data Structures: Pseudocode to C Solutions
Uncover the elegance and efficiency of various data structures through the lens of pseudocode and potential C implementations. The book walks through the design and analysis of algorithms, using pseudocode to represent their core logic. It then connects these pseudocode descriptions to how they would be realized in C, reinforcing practical application.

6. Structured Data with Pseudocode and C Syntax Insights
Dive into the world of structured data management, learning about arrays, linked lists, stacks, queues, and more. The book uses a pseudocode approach to explain the underlying mechanisms of these structures, highlighting their common operations. It also offers insights into C syntax and concepts that are crucial for their actual implementation.

7. Foundational Data Structures: Pseudocode Explained in a C Context
This book provides a comprehensive introduction to fundamental data structures, emphasizing a pseudocode methodology for clear explanation. It explores the principles behind each structure, demonstrating their behavior and performance characteristics. The discussions are framed within a context that relates closely to C programming practices and paradigms.

8. Pseudocode for Data Structure Mastery with C Underpinnings
Achieve mastery of data structures by engaging with a pseudocode-centric learning experience. This title meticulously explains each data structure's logic and operational flow using pseudocode, building a strong conceptual foundation. The underlying C language principles are subtly woven in, providing a robust understanding for potential C programmers.

9. Bridging Algorithms and C: A Pseudocode Approach to Data Structures
Effectively bridge the gap between abstract algorithms and their practical implementation in C by mastering data structures. This book uses pseudocode as the primary tool to dissect and understand data organization, making complex algorithms digestible. It provides clear pathways to

understanding how these pseudocode concepts translate into efficient C code.

Data Structures A Pseudocode Approach With C

Related Articles

- counting atoms in simple molecules with coefficients answer key
- cs lewis reflections on the psalms
- cultural sensitivity training in the workplace

Data Structures A Pseudocode Approach With C

Back to Home: https://www.welcomehomevetsofnj.org