Data Structures and Algorithms (Spring2024)

Week05 (13/14/14-Mar-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Linked List

Definition and Concept

A linked list is a fundamental data structure in computer science, often used as a building block for more complex data structures. A linked list is a collection of elements, called nodes, where each node contains a value and a pointer (or reference) to the next node in the sequence. This structure allows for efficient insertion and deletion of elements as it does not require the elements to be stored contiguously in memory.

Key Characteristics:

- **Dynamic Size:** Unlike arrays, linked lists can grow or shrink in size dynamically, allowing for efficient use of memory.
- **Sequential Access:** Elements in a linked list can be accessed sequentially starting from the first node. There is no direct access to the elements as in arrays.
- **Memory Utilization:** Each node in a linked list requires extra memory for a pointer, in addition to the data it holds.

Types of Linked Lists:

- **Singly Linked List:** Each node has a pointer to the next node.
- **Doubly Linked List:** Each node has pointers to both the next and the previous nodes, allowing for backward traversal.
- **Circular Linked List:** The last node points back to the first node, creating a circular loop.

Comparison with Arrays

Arrays and linked lists are both linear data structures, but they have several key differences that affect their performance and usage in different scenarios.

Memory Allocation:

- **Arrays:** Allocate memory in a contiguous block. This requires knowing the size of the array in advance or may involve costly operations to resize.
- **Linked Lists:** Allocate memory for each element separately, wherever memory is available. This makes linked lists more flexible in terms of memory usage.

Access Time:

- **Arrays:** Provide constant time access (O(1)) to elements using their index, making them efficient for scenarios requiring frequent access to elements by their position.
- **Linked Lists:** Require O(n) time to access the n-th element as it involves traversing the list from the beginning, making them less efficient for direct element access.

Insertion and Deletion:

- **Arrays:** Inserting or deleting elements, especially in the middle of the array, can be costly (O(n)) as it requires shifting elements to maintain continuity.
- **Linked Lists:** Can insert or delete nodes in O(1) time if the pointer to the previous node is known, as it only involves changing pointers.

Memory Efficiency:

- **Arrays:** May lead to wasted memory if the array is not fully utilized or require overhead for resizing operations.
- **Linked Lists:** Use memory more efficiently for dynamic data where the size changes over time, but each node requires extra memory for storing the pointer.

Use Cases:

- **Arrays:** Preferred for applications requiring fast access to elements by index, efficient memory usage when the size of the data is known and does not change frequently.
- **Linked Lists:** Ideal for applications requiring frequent insertion and deletion operations, dynamic memory allocation, or when the size of the data is unknown or changes frequently.

In short, the choice between using an array or a linked list depends on the specific requirements of the application, including memory constraints, operations to be performed, and the importance of access time versus insertion/deletion efficiency.

Applications and Advantages of Linked Lists

Linked lists are versatile data structures that are used in various applications due to their dynamic nature and efficient handling of operations like insertion and deletion. Below are some notable applications and advantages of using linked lists:

Applications:

1. **Dynamic Memory Allocation:** Linked lists are used in managing dynamic memory allocation, where the size of the data structure can grow or shrink at runtime without reallocating the entire structure.
2. **Implementing Other Data Structures:** They serve as the underlying data structure for implementing stacks, queues, graphs, and other more complex data structures.
3. **Undo Functionality in Applications:** Linked lists are used to implement undo functionality in applications like word processors, where operations are stored in a list that can be traversed backwards.

4. **Hash Tables:** Linked lists handle collisions in hash tables, where each bucket is a linked list, allowing multiple entries to exist at the same hash value.
5. **Memory Management:** In operating systems, linked lists manage available memory blocks and are used in the implementation of file systems.
6. **Polynomial Arithmetic:** They are used to represent and perform arithmetic on polynomials, where each node represents a term in the polynomial.

Advantages:

1. **Dynamic Size:** Unlike arrays, the size of a linked list can grow or shrink during runtime, which is beneficial for applications with fluctuating data size requirements.
2. **Efficient Operations:** Insertions and deletions are more efficient in linked lists, especially when performed at the beginning or middle, as they generally require only pointer changes.
3. **No Memory Wastage:** Since linked lists allocate memory as needed for each element, they don't reserve more memory than required, unlike static data structures.
4. **Flexibility:** The non-contiguous memory allocation in linked lists offers flexibility that arrays cannot, supporting efficient memory use under fragmentation.
5. **Simplicity in Reversing:** Reversing a linked list is more straightforward and requires less memory than reversing an array.

Types of Linked Lists

Linked lists can be categorized into three main types based on their structure and how the nodes are linked together. Each type has its unique characteristics and use cases.

Singly Linked List

- **Structure:** Each node contains data and a pointer to the next node in the sequence. The last node points to null, indicating the end of the list.
- **Usage:** Singly linked lists are simple and use less memory. They are suitable for simple applications where only forward traversal is required.

Doubly Linked List

- **Structure:** Nodes in a doubly linked list contain data and two pointers: one pointing to the next node and another pointing to the previous node. This allows for bidirectional traversal.
- **Usage:** Doubly linked lists are used in applications that require frequent traversal in both directions, such as navigation through a browser's history or a music playlist.

Circular Linked List

- **Singly Circular Linked List:** Similar to a singly linked list, but the last node points back to the first node, creating a circular structure.
- **Doubly Circular Linked List:** A doubly linked list where the last node points to the first node and the first node points to the last, allowing bidirectional circular traversal.

- **Usage:** Circular linked lists are used in applications requiring cyclic traversals, such as a round-robin scheduler in operating systems or implementing a multiplayer board game that loops over players.

Each type of linked list offers distinct advantages and is chosen based on the requirements of the application, such as the need for bidirectional traversal, memory usage considerations, and the

Comparing Static Arrays, Dynamic Arrays, and Linked Lists

Aspect	Static Arrays	Dynamic Arrays	Singly Linked Lists	Doubly Linked Lists	Circular Linked Lists
Purpose	Store fixed-size sequential collections of elements.	Store variable-size sequential collections of elements.	Store a collection of elements where each element points to the next.	Store elements where each element points to both the next and the previous.	Store elements in a circular manner for continuous looping.
Uniqueness of Application	Efficient for accessing elements by index.	Allows resizing and efficient indexing.	Efficient for scenarios where insertion and deletion operations dominate.	Allows bidirectional traversal, making it suitable for applications requiring reverse navigation.	Used where the data naturally forms a circle, such as round-robin scheduling or a music playlist on repeat.
Complexity: Creation	O(1)	O(1) for initial creation, O(n) for resizing.	O(1) for a new node.	O(1) for a new node.	O(1) for a new node.
Complexity: Insertion	Not applicable (fixed size)	Amortized O(1), O(n) when resizing is needed.	O(1) at the beginning, O(n) at the end or middle (if the previous node is not known).	O(1) if the node to insert before/after is known.	O(1) if inserting at known points, but requires handling to maintain circularity.
Complexity: Deletion	Not applicable (fixed size)	O(n) to shift elements after deletion.	O(1) with direct access to the node before the one to be deleted, otherwise O(n).	O(1) if the node to be deleted is known.	O(1) with direct access, requires handling to maintain circularity.
Complexity: Access	O(1)	O(1)	O(n)	O(n)	O(n), but loops indefinitely.
Complexity: Search	O(n)	O(n)	O(n)	O(n)	O(n), but may loop indefinitely if not careful.
Other Aspects	- Fixed size limits flexibility. - Does not support operations like insertion or deletion.	- Supports dynamic resizing. - More memory efficient compared to linked lists when data size is known.	- Requires extra memory for the pointer. - Not suitable for random access.	- Requires more memory for pointers compared to singly linked lists. - Easier to implement certain operations (e.g., deletion) due to bidirectional traversal.	- Can complicate certain operations due to the need to handle the circular nature. - Especially suitable for applications requiring cyclic access.

Notes:

- **Complexity** is provided in Big O notation, representing the worst-case scenario.
- The actual performance can vary based on specific implementations and the operations performed.
- This table provides a high-level overview and does not cover all possible nuances and special cases.

Each data structure offers unique benefits and comes with its own set of trade-offs, making it better suited for certain applications over others. Understanding these differences is crucial for choosing the most appropriate data structure for a given problem.

Abstract Data Type (ADT)

An Abstract Data Type (ADT) is a theoretical concept in computer science that defines a data type purely by its behavior from the point of view of a user, including the values it holds, the operations permitted on those values, and the types of operations. ADTs are independent of any specific implementation; they focus on what operations are to be performed but not on how these operations will be executed. Common examples of ADTs include Stack, Queue, List, and Map.

Is Linked List an ADT?

Yes, a linked list can be considered an ADT when defined by its characteristics and operations such as insertion, deletion, traversal, and search without specifying how these operations are implemented. The concrete realization of a linked list, such as a singly linked list, doubly linked list, or circular linked list, represents specific implementations of the linked list ADT.

Comparison Table: Arrays, Dynamic Arrays, and Types of Linked Lists in Creating Advanced Data Structures

The table below compares arrays, dynamic arrays, and different types of linked lists in terms of their suitability and common uses in creating more advanced data structures like stacks, queues, and others.

Data Structure	Arrays	Dynamic Arrays	Singly Linked Lists	Doubly Linked Lists	Circular Linked Lists
Stack	Yes	Yes	Yes	Yes	Not Common
Queue	Yes	Yes	Yes	Yes	Yes (Circular Queue)
Deque	Yes	Yes	No	Yes	Yes
Graphs	Yes	Yes	Yes (Adjacency List)	Yes (Adjacency List)	Not Common
Hash Tables	Yes	Yes	Yes (Chaining)	Yes (Chaining)	Not Common
Trees	Yes	Yes	Yes	Yes (for parent/child representation)	Not Common

Data Structure	Arrays	Dynamic Arrays	Singly Linked Lists	Doubly Linked Lists	Circular Linked Lists

Notes:

- **Stacks** and **Queues** are commonly implemented using arrays and linked lists due to their nature of element addition and removal. Dynamic arrays and doubly linked lists provide more flexibility and efficiency in certain scenarios.
- **Deques (Double-Ended Queues)** benefit significantly from doubly linked lists, which allow efficient insertions and deletions at both ends.
- **Graphs** can be represented using arrays (as adjacency matrices) or linked lists (as adjacency lists), with the choice depending on the graph's density and the operations performed.
- **Hash Tables** often use linked lists to handle collisions through chaining, where doubly linked lists can offer advantages in deletion complexity.
- **Trees** are commonly represented with dynamic arrays or linked lists, where linked lists can provide a natural representation for the hierarchical structure.

Choosing the appropriate underlying data structure depends on the specific requirements of the advanced data structure being implemented, such as memory usage, operation complexity, and the need for dynamic resizing.

Representing a Linked List

In C++, a linked list is typically represented using a structure (struct) that contains at least two components: a data component to hold the actual data, and a pointer component to point to the next node in the list. Here's a basic example of how you might define a linked list node using a struct:

```
#include <iostream>

// Define the structure for a node in the linked list
struct Node {
    int data;       // Data stored in the node
    Node* next;     // Pointer to the next node in the list
};
```

```
int main() {
    // Example of creating a linked list node
    Node* newNode = new Node; // Allocate memory for a new node
    newNode->data = 10;       // Assign a value to the data component
    newNode->next = nullptr;  // Initialize the next pointer to null
(end of list)

    // Output the data stored in the node
    std::cout << "Data in the node: " << newNode->data << std::endl;

    // Deallocate memory for the node
    delete newNode;

    return 0;
}
```

In this example:

- The `Node` struct contains two members: an integer `data` member to store the data and a `Node* next` member, which is a pointer to the next node in the linked list.
- In the `main()` function, memory is allocated for a new node using the `new` keyword.
- The `data` member of the node is assigned a value.
- The `next` pointer is initialized to `nullptr`, indicating that this node is currently the last node in the list.
- After using the node, memory is deallocated using the `delete` keyword to avoid memory leaks.

This is the basic building block for creating a linked list in C++. You can create more nodes and link them together by setting the `next` pointer of one node to point to another node.

Traversing a Linked List

Traversal of a singly linked list involves visiting each node of the list one by one, starting from the head node (the first node) and proceeding to the last node. Here's how traversal can be done on a singly linked list:

```
#include <iostream>

// Define the structure for a node in the linked list
struct Node {
    int data;       // Data stored in the node
    Node* next;     // Pointer to the next node in the list
};
```

```
// Function to traverse the linked list and print its elements
void traverseLinkedList(Node* head) {
    // Start from the head node
    Node* current = head;

    // Traverse the list until reaching the end (nullptr)
    while (current != nullptr) {
        // Output the data stored in the current node
        std::cout << current->data << " ";

        // Move to the next node
        current = current->next;
    }
    std::cout << std::endl;
}



int main() {
    // Example of creating a linked list
    Node* head = new Node;      // Head node
    head->data = 10;            // Assign a value to the data componen
t
    head->next = nullptr;       // Initialize the next pointer to null
(end of list)

    // Create additional nodes
    Node* second = new Node;
    second->data = 20;
    second->next = nullptr;

    Node* third = new Node;
    third->data = 30;
    third->next = nullptr;
```

```
    // Link the nodes together
    head->next = second;
    second->next = third;

    // Traverse the linked list and print its elements
    traverseLinkedList(head);

    // Clean up memory by deallocating nodes
    delete head;
    delete second;
    delete third;

    return 0;
}
```

In this example:

- The `traverseLinkedList()` function takes the head of the linked list as its argument.
- Inside the function, a pointer `current` is initialized to the head node.
- It then iterates over the list using a while loop. At each iteration, it prints the data stored in the current node and moves `current` to the next node using `current = current->next`.
- The loop continues until `current` becomes `nullptr`, indicating the end of the list.
- In the `main()` function, a linked list is created with three nodes, and then the `traverseLinkedList()` function is called to print its elements.

This is how traversal is typically performed on a singly linked list in C++.

Insertion in a Linked List

Insertion in a singly linked list involves adding a new node at a specific position in the list. There are different cases to consider when performing an insertion:

1. Insertion at the beginning of the list.
2. Insertion at the end of the list.
3. Insertion at any position in the middle of the list.

Let's go through each case:

Insertion at the beginning of the list:

```
void insertAtBeginning(Node*& head, int newData) {
    // Create a new node
    Node* newNode = new Node;
    newNode->data = newData;
    // Point the new node to the current head
```

Insertion at the end of the list:

```
void insertAtEnd(Node*& head, int newData) {
    // Create a new node
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    // If the list is empty, make the new node the head
    if (head == nullptr) {
        head = newNode;
        return;
    }



    // Traverse to the end of the list
    Node* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }

    // Insert the new node at the end
    current->next = newNode;
}
```

Insertion at any position in the middle of the list:

```
void insertAtPosition(Node*& head, int position, int newData) {
    // Create a new node
    Node* newNode = new Node;
    newNode->data = newData;
    newNode->next = nullptr;

    // If the position is 0, insert at the beginning
    if (position == 0) {
        newNode->next = head;
        head = newNode;
        return;
    }
```

```
    // Traverse to the node before the position
    Node* current = head;
    for (int i = 0; i < position - 1 && current != nullptr; ++i) {
        current = current->next;
    }

    // Check if the position is valid
    if (current == nullptr) {
        std::cout << "Invalid position\n";
        return;
    }

    // Insert the new node at the specified position
    newNode->next = current->next;
    current->next = newNode;
}
```

These functions provide the basic operations for inserting nodes into a singly linked list. You can choose the appropriate function based on where you want to insert the new node. Remember to handle edge cases such as an empty list or invalid positions accordingly.

Reflection MCQs

1. **What is a linked list?**
    - A) A data structure where elements are stored in contiguous memory locations.
    - B) A collection of elements stored in non-contiguous memory locations, linked using pointers.
    - C) A fixed-size collection of elements.
    - D) A dynamic array with automatic resizing.
    - **Correct Option:** B
2. **How does a singly linked list differ from a doubly linked list?**
    - A) It uses more memory.
    - B) Each node points to the next and previous nodes.
    - C) It only allows forward traversal.
    - D) It cannot store data.
    - **Correct Option:** C

3. **What is the time complexity of accessing an element by index in a linked list?**
    - A) O(1)
    - B) O(log n)
    - C) O(n)
    - D) O(n^2)
    - **Correct Option:** C

4. **Which of the following operations is more efficient in a linked list compared to a dynamic array?**

   - A) Accessing an element by index.
   - B) Insertion at the end.
   - C) Insertion at the beginning.
   - D) Searching for an element.
   - **Correct Option:** C

5. **In a singly linked list, how is the end of the list indicated?**

   - A) By a special end node.
   - B) By a NULL pointer.
   - C) By the largest value.
   - D) By looping back to the first node.
   - **Correct Option:** B

6. **What is an Abstract Data Type (ADT)?**

   - A) A specific implementation of a data structure in a programming language.
   - B) A low-level data type provided by a programming language.
   - C) A type of data structure that allows dynamic resizing.
   - D) A model for data types where the data type is defined by its behavior.
   - **Correct Option:** D

7. **Can arrays be considered an ADT?**

   - A) Yes, because they abstract away memory management.
   - B) No, because they are concrete implementations with fixed operations.
   - C) Yes, because they can grow and shrink dynamically.
   - D) No, because they are a primitive data type.
   - **Correct Option:** B

8. **Which data structure would be most suitable for implementing a browser's back button functionality?**

   - A) Static Array
   - B) Circular Linked List
   - C) Singly Linked List
   - D) Doubly Linked List
   - **Correct Option:** D

9. **What is the time complexity of inserting a new node at the beginning of a singly linked list?**

   - A) O(1)
   - B) O(log n)
   - C) O(n)
   - D) O(n^2)
   - **Correct Option:** A

10. **Which of the following is NOT a correct way to represent a node in a singly linked list using `struct` in C++?**

    - A) `struct Node { int data; Node* next; };`
    - B) `struct Node { int data; Node next; };`
    - C) `struct Node { int data; Node* next = nullptr; };`
    - D) `struct Node { int data; Node* next; Node(int x) : data(x), next(nullptr) {} };`

11. **For traversing a singly linked list to its last node, which of the following is true?**

    - A) You need to know the size of the list beforehand.
    - B) You traverse until you find a node where `node->next` is NULL.
    - C) You can directly access it in O(1) time.
    - D) Traversal is not possible in singly linked lists.
    - **Correct Option:** B

12. **In a singly linked list, what does the `next` pointer of the last node point to?**

    - A) The first node of the list.
    - B) A random node in the list.
    - C) It does not have a `next` pointer.
    - D) NULL.
    - **Correct Option:** D

13. **How do dynamic arrays handle resizing?**

    - A) They automatically delete excess capacity.
    - B) They maintain a fixed size regardless of the number of elements.
    - C) They allocate new memory with larger capacity and copy elements over.
    - D) They compress the existing elements to fit the new data.
    - **Correct Option:** C

14. **Which of the following best describes a circular linked list?**

    - A) A linked list where the `next` pointer of the last node points to NULL.
    - B) A linked list where each node has two pointers: one to the next node and one to the previous node.
    - C) A linked list where the `next` pointer of the last node points to the first node.
    - D) A linked list with nodes in a random order.
    - **Correct Option:** C

15. **What advantage does a doubly linked list have over a singly linked list?**

    - A) It uses less memory per node.
    - B) It allows for direct access to elements by index.
    - C) It allows traversal in both directions.
    - D) It has a fixed size.
    - **Correct Option:** C

16. **What is the primary use of the `NULL` pointer in linked lists?**

- A) To add new nodes to the list.
- B) To indicate the end of the list.
- C) To store data within the nodes.
- D) To connect the list in a circular manner.
- **Correct Option:** B

17. **Which operation is typically more efficient in a linked list compared to an array?**
    - A) Accessing elements by index.
    - B) Random access to elements.
    - C) Insertion at the start of the data structure.
    - D) Searching for an element by its value.
    - **Correct Option:** C
18. **Why might a circular linked list be preferred in a scenario where the data needs to be accessed in a round-robin manner?**
    - A) Because it can store an infinite number of elements.
    - B) Because it allows easier reversal of the list.
    - C) Because it enables continuous looping through the elements without needing to reset to the beginning manually.
    - D) Because it automatically sorts the elements.
    - **Correct Option:** C
19. **What is a potential disadvantage of using a singly linked list?**
    - A) It can only store integer data types.
    - B) It cannot be used to implement stacks or queues.
    - C) It requires more memory due to the storage of pointers.
    - D) It allows for fast random access to elements.
    - **Correct Option:** C
20. **When inserting a new node at the end of a singly linked list, if you do not maintain a tail pointer, what is the time complexity?**
    - A) O(1)
    - B) O(log n)
    - C) O(n)
    - D) O(n^2)
    - **Correct Option:** C

Coding Exercise: Implementing a Singly Linked List in C++

Objective: In this exercise, you will implement a basic singly linked list using `struct` in C++. You will create functions for inserting a node at the beginning, inserting a node at the end, displaying the list, and searching for an element in the list.

Step 1: Define the Node Structure

Start by defining the structure for a list node. Each node should contain an integer data field and a pointer to the next node.

```
struct Node {
    int data;
    Node* next;
};
```

Step 2: Inserting at the Beginning

Implement a function that inserts a new node at the beginning of the list. This function should take a pointer to the head of the list and the data to be inserted as parameters.

```
void insertAtBeginning(Node** head, int newData) {
    // Your code here
}
```

Step 3: Inserting at the End

Implement a function that inserts a new node at the end of the list. This function should also take a pointer to the head of the list and the data to be inserted.

```
void insertAtEnd(Node** head, int newData) {
    // Your code here
}
```

Step 4: Displaying the List

Create a function to print the elements of the list. Traverse from the head to the end, printing each element's data.

```
void displayList(Node* node) {
    // Your code here
}
```

Step 5: Searching for an Element

Implement a function that searches for an element in the list. It should return true if the element is found and false otherwise.

```
bool search(Node* head, int key) {
    // Your code here
```

Instructions:

1. **Implement `insertAtBeginning`:** Create a new node with the given data, point it to the current head, and update the head of the list.
2. **Implement `insertAtEnd`:** If the list is empty, the new node becomes the head. Otherwise, traverse to the end of the list and link the new node.
3. **Complete `displayList`:** Traverse the list from the head while printing the data of each node until the end is reached.
4. **Fill in `search`:** Traverse the list. If a node with the given data is found, return true. If the end of the list is reached, return false.

Sample Main Function:

```
int main() {
    Node* head = nullptr; // Start with an empty list

    // Insert elements
    insertAtBeginning(&head, 10);
    insertAtEnd(&head, 20);
    insertAtBeginning(&head, 5);
    displayList(head);

    // Search for elements
    if(search(head, 20)) {
        std::cout << "Element found." << std::endl;
    } else {
        std::cout << "Element not found." << std::endl;
    }

    return 0;
}
```

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required programs**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind