Data Structures (1.4.2)

List: Dynamic, Mutable, Ordered collection of items, Items can be changed or replaced, Can store more than one data type. Array: Static, Mutable, Ordered collection of items, Items can be changed or replaced, Can only store one data type. Tuple: Static, Immutable, Unordered collection of items, Items cannot be changed or replaced, Can store more than one data type. Static: its size cannot be changed at runtime. Dynamic: its size can be changed at runtime. Immutable: the data cannot be changed at runtime. Mutable: The data can be changed at runtime.

Record: Can be used to store multiple pieces of data that relate to the same thing. Each piece of data can have a different data type. Primitive data type: Built into a programming language. Composite data type: Constructed using other primitive and composite data types.

Linked List: Dynamic. Constructed from nodes and pointers. A start pointer identifies the first node. Each node contains data and a pointer to the next node. By adding an extra pointer, nodes can point to the previous and next items, known as a doubly linked list. A circular linked list can be created by making the last node point to the first node. Each node in a circular linked list can also have an additional pointer pointing to the previous item, creating a doubly circular linked list. Data is stored in order. Advantage: Any available memory address can store data.

Adding a Node:

- Check if there is free memory
- If not, report an error
- If there is, insert the data at the position indicated by the Free pointer
- If the linked list is empty, make our new node the first item (create a start pointer to it) and stop
- Otherwise, start at the first node, proceed through the list linearly, compare each item with the value of the new node, until we find the correct position, or reach the end of the list.
- The new node must point where the previous node pointed.
- The previous node must point to the new node
- Update the free pointer, so that it points to the next available free space

Removing a Node:

- Check if the list is empty, if it is, report an error
- If it is not, check whether the first item is the one to delete
- If it is, update the start pointer to point to the next node
- Otherwise, start at the first node, proceed through the list linearly, compare each item with the value of the new node, until we find the correct node, or reach the end of the list
- Once found, Update the previous node's pointer to point to the next node.
- Update the free pointer

Traverse the List:

- Check if the list is empty, if it is, report an error
- Start at the node pointed to by the Start pointer, output its contents
- Follow the pointers from node to node, outputting as you go

Advantages:

- Dynamic Data structure
- Can be used to implement other data structures (stacks, queues)
- Adding and Deleting is much easier, no need to shift elements afterwards

Disadvantages:

- More memory is needed (to store pointers)
- Direct access is impossible, therefore, traversal takes longer

A stack: A LIFO data structure. Stack pointer points to the node at the top. Pushing onto a full stack leads to a stack overflow. Popping from an empty stack leads to a stack underflow.

Push: Adding an item to the top of the stack

Data Structures (1.4.2)

Pop: Removing an item from the top of the stack
Peek: Returning the value from the top of the stack without removing it

Queue: A FIFO structure. The back pointer points to the last item in the queue. The front pointer points to the first item. Enqueuing onto a full queue leads to a queue overflow. Dequeuing from an empty queue leads to a queue underflow. Cycle the back pointer to the front of the array when it reaches the end (a circular queue). Cycle the front pointer too when necessary.

Enqueue: Adding an item to the back of the queue
Dequeue: Removing an item from the front of the queue
Peek: Returning the value from the front of the queue without removing it

In a linear queue, if your queue reaches the end of the array, you cannot add any more items. With a circular queue, you can add more items if there is space at the front of the array, until the array is full. Priority Queue: each item has a priority. When new items are added, they are inserted ahead of those of lower priority and behind elements of equal and higher priority.

Tree: A fully connected, undirected graph with no cycles. Consists of nodes and pointers. Has a root node at the top. Leaf Node: A node at the bottom of a branch. Nodes are connected with pointers / edges. A subtree: A set of nodes and edges from any single node down through all of its descendants. A binary tree: Rooted Tree, where each node can only have 0, 1 or 2 child nodes. The pointer is set to *null* if there's no child node. Advantage: The time it takes to find a particular item in the tree is less than or equal to the time taken to search through the same data in an array

Pre-order Traversal: Duplicate the binary tree; Node, Left, Right. In-order Traversal: Read the data in order; Left, Node, Right. Post-order Traversal: Delete the binary tree (delete children before parents); Left, Right, Node.

Delete Item

- No children: Delete the node
- One child: Replace the node with its child node, Delete the Child node
- Two children: Replace the node with its in-order successor, Delete the in-order successor

Add Item

- Compare the item to the root node
- If the item is smaller than the root node, go left, Else, go right
- Continue until you reach the bottom of a branch, Place the item here

Hashing Function: Calculates a hash value. Hash Value: The position of an item in a hash table. Two data items cannot occupy the same position in the hash table. If this occurs, this is a collision. Hash table: Data is stored as a collection of key-value pairs. Advantages: locating data is faster and more efficient. A good hashing function should be calculated quickly, result in as few collisions as possible and use as little memory as possible.

Open Addressing: Repeatedly check the next available space in the hash table until an empty position is found and store the item there. Linear Probing: To find the item later, the hashing function delivers the start position from which a linear search can be applied until the item is found. Prevents other items from being stored at their correct locations. Results in Clustering: several positions being filled around common collision values. Rehashing: The process of finding an alternative position for items in the hash table. Chaining: Use a two-dimensional hash table, now more than one item can be placed at the same position.

Graph: Represents complex relationships between objects. Consists of nodes/vertices and pointers/edges. Each vertex can have more than two edges and point to any vertex in the data structure. The edges can either point in one direction (directed) or not specify a direction (undirected). Each edge can have a weight associated with it, representing something about the relationship between the vertices (weighted graph). Adjacency List: Graph stored as a dictionary. Adjacency Matrix: Graph stored as an array. A tree is a type of graph.