Data Structures

Veri Yapıları

Arrays

- The array is the most commonly used data storage structure; it's built into most programming languages.
- Because arrays are so well known, they offer a convenient jumpingoff place for introducing data structures and for seeing how object-oriented programming and data structures relate to one another.
- In many situations the array is the first kind of structure you should consider when storing and manipulating data.

Arrays

- Arrays are useful when
- • The amount of data is reasonably small.
- • The amount of data is predictable in advance.
- If you have plenty of memory, you can relax the second condition; just make the array big enough to handle any foreseeable influx of data.

Simple Sorting

- As soon as you create a significant database, you'll probably think of reasons to sort it in various ways.
- You need to arrange names in alphabetical order, students by grade, customers by ZIP code, home sales by price, cities in order of increasing population, countries by GNP, stars by magnitude, and so on
- Because sorting is so important and potentially so timeconsuming, it has been the subject of extensive research in computer science, and some very sophisticated methods have been developed.

Stacks

- A stack allows access to only one data item: the last item inserted.
- If you remove this item, you can access the next-to-last item inserted, and so on.
- This capability is useful in many programming situations.
- A stack is also a handy aid for algorithms applied to certain complex data structures.

Stacks

- Most microprocessors use a stack-based architecture.
- When a method is called, its return address and arguments are pushed onto a stack, and when it returns, they're popped off.
- The stack operations are built into the microprocessor.
- Some older pocket calculators used a stack-based architecture.
- Instead of entering arithmetic expressions using parentheses, you pushed intermediate results onto a stack.

Stacks

- A stack is used when you want access only to the last data item inserted; it's a LastIn-First-Out (LIFO) structure.
- A stack is often implemented as an array or a linked list.
- The array implementation is efficient because the most recently inserted item is placed at the end of the array, where it's also easy to delete.
- Stack overflow can occur, but is not likely if the array is reasonably sized, because stacks seldom contain huge amounts of data.

Stacks

- If the stack will contain a lot of data and the amount can't be predicted accurately in advance (as when recursion is implemented as a stack), a linked list is a better choice than an array.
- A linked list is efficient because items can be inserted and deleted quickly from the head of the list.
- Stack overflow can't occur (unless the entire memory is full).
- A linked list is slightly slower than an array because memory allocation is necessary to create a new link for insertion, and deallocation of the link is necessary at some point following removal of an item from the list

Queues

- The word queue is British for line (the kind you wait in).
- In Britain, to “queue up” means to get in line.
- In computer science a queue is a data structure that is somewhat like a stack, except that in a queue the first item inserted is the first to be removed (First-In-First-Out, FIFO), while in a stack, as we’ve seen, the last item inserted is the first to be removed (LIFO).
- A queue works like the line at the movies:
- The first person to join the rear of the line is the first person to reach the front of the line and buy a ticket.
- The last person to line up is the last person to buy a ticket (or—if the show is sold out—to fail to buy a ticket).

Queues

- Queues are used as a programmer's tool as stacks are.
- There are various queues quietly doing their job in your computer's (or the network's) operating system.
- There's a printer queue where print jobs wait for the printer to be available.
- A queue also stores keystroke data as you type at the keyboard.
- This way, if you're using a word processor but the computer is briefly doing something else when you hit a key, the keystroke won't be lost; it waits in the queue until the word processor has time to read it.
- Using a queue guarantees the keystrokes stay in order until they can be processed.

Queues

- A queue is used when you want access only to the first data item inserted; it's a First-In-First-Out (FIFO) structure.
- Like stacks, queues can be implemented as arrays or linked lists.
- Both are efficient.
- The array requires additional programming to handle the situation in which the queue wraps around at the end of the array.

Queues

- A linked list must be double-ended, to allow insertions at one end and deletions at the other.
- As with stacks, the choice between an array implementation and a linked list implementation is determined by how well the amount of data can be predicted.
- Use the array if you know about how much data there will be; otherwise, use a linked list.

Linked Lists

- The linked list is a versatile mechanism suitable for use in many kinds of general-purpose databases.
- It can also replace an array as the basis for other storage structures such as stacks and queues.
- In fact, you can use a linked list in many cases in which you use an array, unless you need frequent random access to individual items using an index.
- Linked lists aren’t the solution to all data storage problems, but they are surprisingly versatile and conceptually simpler than some other popular structures such as trees.

Linked Lists

- Consider a linked list whenever the amount of data to be stored cannot be predicted in advance or when data will frequently be inserted and deleted.
- The linked list obtains whatever storage it needs as new items are added, so it can expand to fill all of available memory; and there is no need to fill “holes” during deletion, as there is in arrays.

Linked Lists

- Insertion is fast in an unordered list.
- Searching and deletion are slow (although deletion is faster than in an array), so, like arrays, linked lists are best used when the amount of data is comparatively small.
- A linked list is somewhat more complicated to program than an array, but is simple compared with a tree or hash table.

Recursion

- Recursion is a programming technique in which a method (function) calls itself.
- This may sound like a strange thing to do, or even a catastrophic mistake.
- Recursion is, however, one of the most interesting, and one of the most surprisingly effective, techniques in programming.
- However, it not only works, it also provides a unique conceptual framework for solving many problems

Shellsort

- The Shellsort is named for Donald L. Shell, the computer scientist who discovered it in 1959.
- It's based on the insertion sort, but adds a new feature that dramatically improves the insertion sort's performance.
- The Shellsort is good for medium-sized arrays, perhaps up to a few thousand items, depending on the particular implementation.
- It's not quite as fast as quicksort and other O(N*logN) sorts, so it's not optimum for very large files.

Shellsort

- However, it’s much faster than the O(N2) sorts like the selection sort and the insertion sort, and it’s very easy to implement:
- The code is short and simple.
- The worst-case performance is not significantly worse than the average performance.
- Some experts recommend starting with a Shellsort for almost any sorting project and changing to a more advanced sort, like quicksort, only if Shellsort proves too slow in practice.

Quick Sort

- Quicksort is undoubtedly the most popular sorting algorithm, and for good reason: In the majority of situations, it's the fastest, operating in O(N*logN) time.
- (This is only true for internal or in-memory sorting; for sorting data in disk files, other algorithms may be better.)
- Quicksort was discovered by C.A.R. Hoare in 1962.
- Basically, the quicksort algorithm operates by partitioning an array into two subarrays and then calling itself recursively to quicksort each of these subarrays.

Radix Sort

- The radix sort disassembles the key into digits and arranges the data items according to the value of the digits.
- Amazingly, no comparisons are necessary.

Binary Trees

- Binary trees are one of the fundamental data storage structures used in programming.
- Why might you want to use a tree?
- Usually, because it combines the advantages of two other structures: an ordered array and a linked list.
- You can search a tree quickly, as you can an ordered array, and you can also insert and delete items quickly, as you can with a linked list.

Binary Trees

- Imagine an array in which all the elements are arranged in order—that is, an ordered array.
- You can quickly search such an array for a particular value, using a binary search.
- You check in the center of the array; if the object you're looking for is greater than what you find there, you narrow your search to the top half of the array; if it's less, you narrow your search to the bottom half.
- Applying this process repeatedly finds the object in O(logN) time.
- You can also quickly iterate through an ordered array, visiting each object in sorted order.

Binary Trees

- On the other hand, if you want to insert a new object into an ordered array, you first need to find where the object will go, and then move all the objects with greater keys up one space in the array to make room for it.
- These multiple moves are time-consuming, requiring, on the average, moving half the items (N/2 moves).
- Deletion involves the same multimove operation and is thus equally slow.
- If you’re going to be doing a lot of insertions and deletions, an ordered array is a bad choice.

Binary Trees

- If every node in a tree can have at most two children, the tree is called a binary tree.
- The two children of each node in a binary tree are called the left child and the right child, corresponding to their positions when you draw a picture of a tree.
- A node in a binary tree doesn't necessarily have the maximum of two children; it may have only a left child, or only a right child, or it can have no children at all (in which case it's a leaf).
- A binary tree is the first structure to consider when arrays and linked lists prove too slow.
- A tree provides fast O(logN) insertion, searching, and deletion.

Red-Black Trees

- “Binary Trees,” ordinary binary search trees offer important advantages as data storage devices: You can quickly search for an item with a given key, and you can also quickly insert or delete an item.
- Other data storage structures, such as arrays, sorted arrays, and linked lists, perform one or the other of these activities slowly.
- Thus, binary search trees might appear to be the ideal data storage structure.
- Unfortunately, ordinary binary search trees suffer from a troublesome problem.
- They work well if the data is inserted into the tree in random order.

Red-Black Trees

- However, they become much slower if data is inserted in already-sorted order (17, 21, 28, 36,…) or inversely sorted order (36, 28, 21, 17,…).
- When the values to be inserted are already ordered, a binary tree becomes unbalanced.
- With an unbalanced tree, the ability to quickly find (or insert or delete) a given element is lost.
- The red-black tree is a binary search tree with some added features.
- There are other ways to ensure that trees are balanced.

Hash Tables

- A hash table is a data structure that offers very fast insertion and searching.
- When you first hear about them, hash tables sound almost too good to be true.
- No matter how many data items there are, insertion and searching (and sometimes deletion) can take close to constant time: O(1) in big O notation.
- In practice this is just a few machine instructions.
- For a human user of a hash table, this is essentially instantaneous.

Hash Tables

- It’s so fast that computer programs typically use hash tables when they need to look up tens of thousands of items in less than a second (as in spelling checkers).
- Hash tables are significantly faster than trees, operate in relatively fast O(logN) time.
- Not only are they fast, hash tables are relatively easy to program.
- Hash tables do have several disadvantages.
- They’re based on arrays, and arrays are difficult to expand after they’ve been created.

Hash Tables

- For some kinds of hash tables, performance may degrade catastrophically when a table becomes too full, so the programmer needs to have a fairly accurate idea of how many data items will need to be stored.
- Also, there's no convenient way to visit the items in a hash table in any kind of order (such as from smallest to largest).
- If you need this capability, you'll need to look elsewhere.
- However, if you don't need to visit items in order, and you can predict in advance the size of your database, hash tables are unparalleled in speed and convenience.
- Hash tables are the fastest data storage structure.

Hash Tables

- This makes them a necessity for situations in which a computer program, rather than a human, is interacting with the data.
- Hash tables are typically used in spelling checkers and as symbol tables in computer language compilers, where a program must check thousands of words or symbols in a fraction of a second.
- Hash tables may also be useful when a person, as opposed to a computer, initiates data-access operations.
- Hash tables are not sensitive to the order in which data is inserted, and so can take the place of a balanced tree.
- Programming is much simpler than for balanced trees.

Hash Tables

- Hash tables require additional memory, especially for open addressing.
- Also, the amount of data to be stored must be known fairly accurately in advance, because an array is used as the underlying structure.
- A hash table with separate chaining is the most robust implementation, unless the amount of data is known accurately in advance, in which case open addressing offers simpler programming because no linked list class is required.
- Hash tables don’t support any kind of ordered traversal, or access to the minimum or maximum items.
- If these capabilities are important, the binary search tree is a better choice.

Heaps

- A heap is a kind of tree.
- It offers both insertion and deletion in O(logN) time.
- Thus, it's not quite as fast for deletion, but much faster for insertion.
- It's the method of choice for implementing priority queues where speed is important and there will be many insertions.

Heaps

- A heap is a binary tree with these characteristics:
- • It’s complete.
- • It’s (usually) implemented as an array.
- • Each node in a heap satisfies the heap condition, which states that every node’s key is larger than (or equal to) the keys of its children.

Graphs

- Graphs are data structures rather like trees.
- In fact, in a mathematical sense, a tree is a kind of graph.
- In computer programming, however, graphs are used in different ways than trees.
- For example, a binary tree is shaped the way it is because that shape makes it easy to search for data and insert new data.
- The edges in a tree represent quick ways to get from node to node.

Graphs

- Graphs, on the other hand, often have a shape dictated by a physical or abstract problem.
- For example, nodes in a graph may represent cities, while edges may represent airline flight routes between the cities.
- Another more abstract example is a graph representing the individual tasks necessary to complete a project.
- In the graph, nodes may represent tasks, while directed edges (with an arrow at one end) indicate which task must be completed before another.
- In both cases, the shape of the graph arises from the specific realworld situation.

Graphs

- Before going further, we must mention that, when discussing graphs, nodes are traditionally called vertices (the singular is vertex).
- This is probably because the nomenclature for graphs is older than that for trees, having arisen in mathematics centuries ago.
- Trees are more closely associated with computer science.
- However, both terms are used more or less interchangeably
- Graphs are unique in the pantheon of data storage structures.

Graphs

- They don’t store general-purpose data, and they don’t act as programmer’s tools for use in other algorithms.
- Instead, they directly model real-world situations.
- The structure of the graph reflects the structure of the problem.
- When you need a graph, nothing else will do, so there’s no decision to be made about when to use one.
- The primary choice is how to represent the graph: using an adjacency matrix or adjacency lists.

Graphs

- Your choice depends on whether the graph is full, when the adjacency matrix is preferred, or sparse, when the adjacency list should be used.
- The depth-first search and breadth-first search run in O(V2) time, where V is the number of vertices, for adjacency matrix representation.
- They run in O(V+E) time, where E is the number of edges, for adjacency list representation.

Graphs

- Minimum spanning trees and shortest paths run in O(V2) time using an adjacency matrix and O((E+V)logV) time using adjacency lists.
- You'll need to estimate V and E for your graph and do the arithmetic to see which representation is appropriate.