Chapter 3. Data Structures

Larry | Peng Yang · Follow

14 min read · Sep 4, 2018

3.1 Contiguous vs. Linked Data Structures

Data structures can be neatly classified as either ***contiguous*** or ***linked***, depending upon whether they are based on arrays or pointers:

- ***Contiguously-allocated structures*** are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- ***Linked data structures*** are composed of distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists.

Arrays

- ***Constant-time access given the index.*** O(1) to read
- ***Space efficiency.*** No links or other formatting information, no end-of-record information as the size is fixed.
- ***Memory locality.*** Arrays are good for iterating all the elements of a data structure because they exhibit excellent memory locality. Physical continuity between successive data accesses helps exploit the ***high-speed cache memory*** on modern computer architectures.

> ***Cache memory****, also called CPU memory, is high-speed static random access memory (SRAM) that a computer microprocessor can access more quickly than it can access regular random access memory (RAM). This memory is typically integrated directly into the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU.*
> *https://searchstorage.techtarget.com/definition/cache-memory*

The downside of arrays is that we cannot adjust their size in the middle of a program's execution. Actually, we can efficiently enlarge arrays as we need them, through the miracle of dynamic arrays. Suppose we start with an array of size 1, and double its size from m to 2m each time we run out of space.

How many times might an element have to be recopied after a total of n insertions? Well, the first inserted element will have been recopied when the array expands after the first, second, fourth, eighth, . . . insertions. ***It will take log2 n doublings until the array gets to have n positions.*** However, most elements do not suffer much upheaval. Indeed, the (n/2 + 1)st through nth elements will move at most once and might never have to move at all. If half the elements move once, a quarter of the elements twice, and so on, the total number of movements M is given as below.

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i \leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

Thus, ***each of the n elements move only two times on average, and the total work of managing the dynamic array is the same O(n)*** as it would have been if a single array of sufficient size had been allocated in advance

> *With the formula above, when n = 8, lg n = 3. M = 1*(8/2) + 2*(8/4) + 3*(8/8) = 4+4+3 = 11 < 2n = 16.*

> *The first term in the sum is 1·n/2, representing the fact that n/2 elements (namely, elements (n/2+1),…,n(n/2+1),…,n) have been copied exactly once (just now!).*
>
> *The next term is 2·n/4. The previous n/4 elements — that is, elements (n/4)+1,…,n/2(n/4)+1,…,n/2 — will have been copied exactly twice, once just now and once when we expanded the array from size n/2 to n.*
>
> *The next term is 3·n/8. The previous n/8 elements will have been copied three times: now, at the last expansion and at the second-to-last expansion when we expanded from size n/4 to n/2. You can probably see where this is going by now.*
>
> *The thing to remember is that each element gets copied every time the array expands* after *the element in question has been inserted, and the copies happen every time the size exceeds a power of two.*
>
> *https://softwareengineering.stackexchange.com/questions/119151/how-many-copies-are-needed-to-enlarge-an-array*

$$M = 1 + 2 + 2^2 + \cdots + 2^{\lfloor \log_2 n \rfloor} = 2^{\lfloor \log_2 n \rfloor + 1} - 1 \leq 2^{1+\log_2 n} - 1 = 2n - 1$$

```
array of 1 element
+--+
|a |
+--+

double the array (2 elements)
+--++--+
|a ||b |
+--++--+

double the array (4 elements)
+--++--++--++--+
|a ||b ||c ||c |
+--++--++--++--+

double the array (8 elements)
+--++--++--++--++--++--++--++--+
|a ||b ||c ||c ||x ||x ||x ||x |
+--++--++--++--++--++--++--++--+

double the array (16 elements)
+--++--++--++--++--++--++--++--++--++--++--++--++--++--++--++--+
|a ||b ||c ||c ||x ||x ||x ||  ||  ||  ||  ||  ||  ||  ||  ||  |
+--++--++--++--++--++--++--++--++--++--++--++--++--++--++--++--+
```

Pointers and Linked Structures

Pointers in C have types declared at compiler time, denoting the data type of the items they can point to.

The list is the simplest linked structure. The three basic operations supported by lists are ***searching***, ***insertion***, and ***deletion.***

- *Searching*: Can be done iteratively or recursively.
- *Insertion*: Generally to be done iteratively. A lot of variations, such as insert before the head, insert at nth position or insert at end.
- *Deletion*: Can be done literately or recursively. Find the predecessor element, point its next element to the next of element to be deleted, delete element. Also need to consider the case that the node to be deleted is the head of the list.

Comparison

The relative advantages of linked lists over static arrays include:

- Overflow on linked structures can never occur unless the memory is actually full.
- Insertions and deletions are simpler than for contiguous (array) lists.
- With large records, moving pointers is easier and faster than moving the items themselves.

while the relative advantages of arrays include:

- Linked structures require extra space for storing pointer fields.

- Linked lists do not allow efficient random access to items.
- Arrays allow better memory locality and cache performance than random pointer jumping.

One final thought about these fundamental structures is that they can be thought of as recursive objects. This insight leads to simpler list processing, and efficient divide-and-conquer algorithms such as quicksort and binary search.

3.2 Stacks and Queues

We use the term ***container*** to denote a data structure that permits storage and retrieval of data items independent of content.

Stacks(LIFO) are simple to implement and very efficient. For this reason, stacks are probably the right container to use when retrieval order doesn't matter at all, such as when processing batch jobs. Algorithmically, LIFO tends to happen in the course of executing recursive algorithms.

Queues(FIFO) are somewhat trickier to implement than stacks and thus are most appropriate for applications (like certain simulations) where the order is important. You want the container holding jobs to be processed in FIFO

order to ***minimize the maximum time spent waiting***. Note that the average waiting time will be the same regardless of whether FIFO or LIFO is used.

Queues are often used as the fundamental data structure to control breadth-first searches in graphs.

Stacks and queues can be effectively implemented using ***either arrays or linked lists***. The key issue is whether an upper bound on the size of the container is known in advance, thus permitting the use of a statically-allocated array.

3.3 Dictionaries

The dictionary data type permits access to data items by content. You stick an item into a dictionary so you can find it when you need it.

The primary operations of dictionary support are:

- ***Search(D,k)*** — Given a search key k, return a pointer to the element in dictionary D whose key value is k, if one exists.
- ***Insert(D,x)*** — Given a data item x, add it to the set in the dictionary D.

- ***Delete(D,x)*** — Given a pointer to a given data item x in the dictionary D, remove it from D.
- ***Max(D) or Min(D)*** — Retrieve the item with the largest (or smallest) key from D. This enables the dictionary to serve as a priority queue.
- ***Predecessor(D,k)*** or ***Successor(D,k)*** — Retrieve the item from D whose key (here, the value in the dictionary) is immediately before (or after) k in sorted order. These enable us to iterate through the elements of the data structure.

Problem: What are the asymptotic worst-case running times for each of the seven fundamental dictionary operations (search, insert, delete, successor, predecessor, minimum, and maximum) when the data structure is implemented as ***unsorted array and sorted array.***

Dictionary operation	Unsorted array	Sorted array
Search(L, k)	$O(n)$	$O(\log n)$
Insert(L, x)	$O(1)$	$O(n)$
Delete(L, x)	$O(1)^*$	$O(n)$
Successor(L, x)	$O(n)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(1)$
Minimum(L)	$O(n)$	$O(1)$
Maximum(L)	$O(n)$	$O(1)$

Unsorted array

> ***Insertion*** *is implemented by incrementing n and then copying item x to the nth cell in the array, A[n]. The bulk of the array is untouched, so this operation takes constant time.*
>
> **Deletion**. *Just write over A[x] with A[n], and decrement n. This only takes constant time.*

> ***Successor and Predecessor****. In an unsorted array an element's physical predecessor (successor) is not necessarily its logical predecessor (successor). Instead, the predecessor of A[x] is the biggest element smaller than A[x]. Similarly, the successor of A[x] is the smallest element larger than A[x]. Both require a sweep through all n elements of A to determine the winner.*

Problem: What is the asymptotic worst-case running times for each of the seven fundamental dictionary operations when the data structure is implemented as

- A singly-linked unsorted list.
- A doubly-linked unsorted list.
- A singly-linked sorted list.
- A doubly-linked sorted list.

Dictionary operation	Singly unsorted	Double unsorted	Singly sorted	Doubly sorted
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Delete(L, x)	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Successor(L, x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Minimum(L)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Maximum(L)	$O(n)$	$O(n)$	$O(1)^*$	$O(1)$

> *For **deletion**, we have to find the predecessor of the element to be deleted which requires O(n) in singly-linked list. **Deletion** is faster for sorted doubly-linked lists (O(1)) than sorted arrays, because splicing out the deleted element from the list is more efficient than filling the hole by moving array elements (O(n)).*
>
> ***Maximum** — The maximum element sits at the tail of the list, which would **normally** require Θ(n) time to reach in either singly- or doubly-linked lists. However, we can maintain a separate pointer to the list tail, provided we pay the maintenance costs for this pointer **on every insertion and deletion**. The tail pointer can be updated in constant time on doubly-linked lists: on insertion check whether **last->next** still equals NULL(as after the insertion, the list should be sorted, if it is NULL, then no need to update the tail pointer, if it is not NULL, then update it to point to its next element which is the last element), and on*

> *deletion set **last** to point to the list predecessor of last if the last element is deleted. For singly-linked link, the trick is to charge the cost to each deletion, which already took linear time.*

3.4 Binary Search Trees

Binary search requires that we have fast access to ***two elements*** — specifically the median elements above and below the given node. To combine these ideas, we need a "linked list" with two pointers per node. This is the basic idea behind binary search trees.

Implementing Binary Search Trees

Binary tree nodes have left and right pointer fields, ***an (optional) parent pointer***, and a data field.

The basic operations supported by binary trees are ***searching***, ***traversal***, ***insertion***, and ***deletion***.

Searching in a Tree. The recursive structure yields the recursive search algorithm below: This search algorithm runs in O(h) time, where h denotes the height of the tree.

```
tree *search_tree(tree *l, item_type x){
    if (l == NULL) return(NULL);
    if (l->item == x) return(l);
    if (x < l->item)
        return( search_tree(l->left, x) );
    else
        return( search_tree(l->right, x) );
}
```

Finding Minimum and Maximum Elements in a Tree. The minimum element must be the leftmost descendent of the root, and the maximum element must be the rightmost descendent of the root.

```
tree *find_minimum(tree *t) {
    tree *min;  //pointer to minimum
    if (t == NULL) return(NULL);
    min = t;
    while (min->left != NULL) //Iteratively
        min = min->left;
    return(min);
}
```

Traversal in a Tree. in-order, pre-order and post-order. Each item is processed once during the course of traversal, which runs in O(n) time, where n denotes the number of nodes in the tree.

```
void traverse_tree(tree *l){ // in-order
    if (l != NULL) {
        traverse_tree(l->left);
        process_item(l->item);
        traverse_tree(l->right);
    }
}
```

Insertion in a BST

- There is ***only one place*** to insert an item x into a binary search tree T where we know we can find it again. We must replace the NULL pointer found in T after an unsuccessful query for the key k.
- This implementation uses recursion to combine the search (O(n))and node insertion (O(1)) stages of key insertion.
- The three arguments to insert tree are (1) a pointer l to the pointer linking the search subtree to the rest of the tree, (2) the key x to be inserted, and (3) a parent pointer to the parent node containing l (if the tree has parent pointer). The node is allocated and linked in on hitting the NULL pointer.
- The newly inserted node ***only has NULL left and NULL right nodes unless it is an AVL tree.***

```
/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL){
       node* p = malloc(sizeof(tree)); /* allocate new node */
       p->key = key;
       p->left = p->right = NULL;
       return p;
    }

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

Deletion from a BST. https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/

The worst-case complexity analysis is as follows. Every deletion requires the cost of at most two search operations, each taking O(h) time where h is the height of the tree, plus a constant amount of pointer manipulation.

There are three cases:

- ***Node to be deleted is leaf:*** Simply remove from the tree.
- ***Node to be deleted has only one child:*** Copy the child to the node and delete the child.
- ***Node to be deleted has two children:*** Find *in-order successor* (smallest value in the right subtree, minValueNode(node->right), node 5 in below figure) of the node (4 in below figure). Copy contents of the in-order successor to the node and delete the in-order successor. Note that in-order predecessor can also be used.

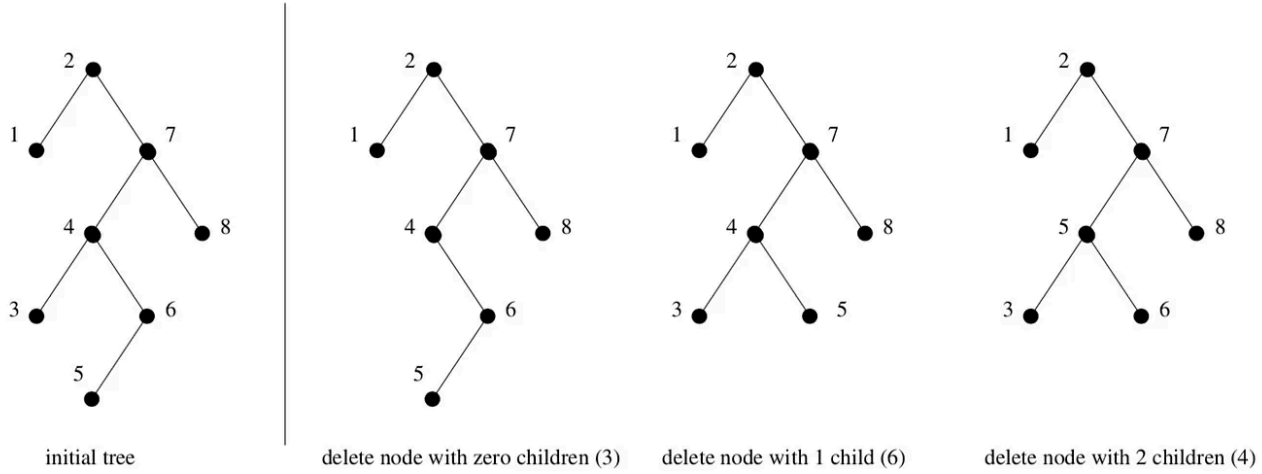

Figure 3.4: Deleting tree nodes with 0, 1, and 2 children

How Good Are Binary Search Trees?

Bad things can happen when building trees through insertion. The data structure has no control over the order of insertion. Consider what happens if the user inserts the keys in sorted order. The operations insert(a), followed by insert(b), insert(c), insert(d), . . . will produce a skinny linear height tree where only right pointers are used. Thus binary trees can have heights *ranging from lg n to n.* There are n! possible insertion orderings, with high probability the resulting tree will have O(log n) height.

3.5 Priority Queues

https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

Priority queues are data structures that provide more flexibility than simple sorting, because they allow new elements to enter a system at arbitrary intervals. It is much more cost-effective to insert a new job into a priority queue than to re-sort everything on each such arrival.

It is an extension of queue with following properties.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.

- If two elements have the same priority, they are served according to their order in the queue.

The basic priority queue supports three primary operations:

- ***Insert***(Q,x)– Given an item x with key k, insert it into the priority queue Q.
- ***Find-Minimum***(Q) or Find-Maximum(Q)– Return a pointer to the item whose key value is smaller (larger) than any other key in the priority queue Q.
- ***Delete-Minimum***(Q) or Delete-Maximum(Q)– Remove the item from the pri- ority queue Q whose key is minimum (maximum).

How to implement priority queue?

- ***Using Array:*** A simple implementation is to use array of following structure. (Linked list can also be used)

```
struct item {
   int item;
   int priority;
}
```

	Unsorted array	Sorted array	Balanced tree
Insert(Q, x)	$O(1)$	$O(n)$	$O(\log n)$
Find-Minimum(Q)	$O(1)$	$O(1)$	$O(1)$
Delete-Minimum(Q)	$O(n)$	$O(1)$	$O(\log n)$

- **Using Heaps:** Heap is generally preferred for priority queue implementation because heaps provide better performance compared arrays or linked list.

3.6 War Story: Stripping Triangulations (Omitted)

3.7 Hashing and Strings

Collision Resolution

- *Chaining* is the easiest approach to collision resolution. Represent the hash table as an array of m linked lists. It requires extra memory to store the pointers.
- The alternative is *open addressing*. The simplest possibility (called sequential probing) inserts the item in the next open spot in the table.

Deletion in an open addressing scheme can get ugly, since removing one element might break a chain of insertions, making some elements inaccessible. We have no alternative but to reinsert all the items in the run following the new hole. Also see this link: https://stackoverflow.com/questions/9127207/hash-table-why-deletion-is-difficult-in-open-addressing-scheme

> *Assume* `hash(x) = hash(y) = hash(z) = i` *. And assume* `x` *was inserted first, then* `y` *and then* `z` *.*
> *In open addressing:* `table[i] = x, table[i+1] = y, table[i+2] = z.`
>
> *Now, assume you want to delete* `x` *, and set it back to* `NULL` *. When later you will search for* `z` *, you will find that* `hash(z) = i` *and* `table[i] = NULL` *, and you will return a wrong answer:* `z` *is not in the table.*
>
> *To overcome this, you need to set* `table[i]` *with a special marker indicating to the search function to keep looking at index* `i+1` *, because there might be element there which its hash is also* `i` *.*

Efficient String Matching via Hashing

Substring Pattern Matching

- The simplest algorithm to search for the presence of pattern string p in text t overlays the pattern string at every position in the text, and checks whether every pattern character matches the corresponding text character, ***O(nm)*** time, where n = |t| and m = |p|(Demonstrated in chapter 2).

- But we can give a linear expected-time algorithm for string matching, called the ***Rabin-Karp algorithm.*** It is based on hashing. Suppose we compute a given hash function on both the pattern string p and the m-character substring starting from the ith position of t. If these two strings are identical, clearly the resulting hash values must be the same. If the two strings are different, the hash values will almost certainly be different (due to collision). These false positives should be so rare that we can easily spend the O(m) (the length of the substring) time it takes to explicitly check the identity of two strings whenever the hash values agree.

```
function RabinKarp(string t[1..n], string p[1..m])
  hpattern := hash(p[1..m]); // length = m-1+1 = m
  for i from 1 to n-m+1 //e.g. n=5, m=3, 1->n-m+1=>1->3. n-m+1-1+1 =
n-m+1 times of hash computations.
    hs := hash(t[i..i+m-1]) // length = i+m-1-i+1 = m
    if hs = hpattern
      if t[i..i+m-1] = p[1..m]
        return i
  return not found
```