Data Structures

June 12, 2020

1 Contiguous vs. Linked List

Contiguously Allocated Structures: > Are composed of single slabs of memory and include arrays, matricies, heaps and hash tables

Linked Data Structure: > Are composed of distinct chunks of memory bound together by pointers and include lists, trees, and graph adjacency list

1.1 Arrays

The array is the fundamental contiguously-allocated data structure

Arrays are fixed size and each element can be located by its index or equivalently adress

Array Advantages: - Constant-time access given the index - because the index of each element maps directly to a particular memory adress, we can access arbitrary data items instantly provided we know the index - Space efficency - Arrays consist purely of data, so no spaced is wasted with links or other formatting information. End of record information is also not needed because we know of the size beforehand - Memory Locality - A common programming idiom involved iterating through all the elements of a data structure. Arrays are good for this because they exhibit excellent memory locality. Physical continuity between successive data accesses helps exploit the cache-memory on modern computers

Dynamic arrays actually allow us to enlarge arrays as we need them. What we do is simply make a new array $ 2X $ the previous arrays size and copy everything over to the new array

Note the doubling involved and thing $ logs $. It will take $ log_2(n) $ doubling untill the arrays get to have $ n $ positions. Of course in reality, each element moves only two times on average

Total work 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!

The primary thing lost using dynamic arrays is guarantee that each array access takes constant time in the *worst case.*

Your basically amortizing the costs. The doubling price is spread over time

1.2 Pointers and Linked Structures

Pointers are the connections that hold the pieces of linked structures together. They represent adress of location in memory. A special NULL pointer value is used to denote structure-terminating or unassigned pointers.

Figure 1: Linked list example showing data and pointer fields

- Each node in our data structure contains one or more data feilds that retain the data that we need to store
- Each node contains a pointer feild to at least one other node. This means that much of the space used in linked data structre has to be devoted to pointers, not data
- Finally, we need a pointer to head of the structure, so we know where to access it

1.2.1 Searching A List

Searching for a item $ x $ in a linked list can be done iteratively or recursively. We do this untill we reach the $ NULL $ or $ 1 $

```
list *search_list(list *l, item_type x)
{
    if (l == NULL) return(NULL);

    if (l->item == x)
        return(l);
    else
        return( search_list(l->next, x) );
    }
```

1.3 List Insertion

Insertion at the beginning of the list avoids any need to traverse the list, but does require us to update the pointer to the head of the data structure

```
void insert_list(list **l, item_type x)
    {
    list *p; /* temporary pointer */

    p = malloc( sizeof(list) );
    p->item = x;
    p->next = *l;
    *l = p;
    }
```

- First, the `malloc` function allocates a chunk of memory of sufficient size for a new node to contain $ x $
- Secondly, the funny double star $ (**1) $ denotes that $ 1 $ is a pointer to a pointer to a list node. The last line $ *1 = p $ copies $ p $ to the place pointed to $ 1 $, which is the external variable maintaining access to the head of the list

1.3.1 List Deletion

Deletion form a linked list:

- find a pointer to predecessor of the item to be deleted
- the predecessor is needed because it points to the doomed node, so its $ next $ pointer must changes.

- the deletion is easy, one the node is found, just take care to reset the pointer to the head of the list $ 1 $ when the first element is deleted

```
list *predecessor_list(list *l, item_type x)
{
    if ((l == NULL) || (l->next == NULL)) {
        printf("Error: predecessor sought on null list.\n");
        return(NULL);
}

if ((l->next)->item == x)
    return(l);
else
    return( predecessor_list(l->next, x) );
}


delete_list(list **l, item_type x)
{
    list *p; /* item pointer */
    list *pred; /* predecessor pointer */
    list *search_list(), *predecessor_list();

    p = search_list(*l,x);
    if (p != NULL) {
        pred = predecessor_list(*l,x);
        if (pred == NULL) /* splice out out list */
            *l = p->next;
        else
            pred->next = p->next;
        free(p); /* free memory used by node */
    }
}
```

1.4 Comparsion

Linked List Pros: - Overflow on linked structures can never occur unless the memory is actually full - Insertion and deletions are simpler than for contigious lists - With large records, moving pointer is easier and faster than moving the items themselves

Static Array Pros: - Linked structures require extra space for storing pointer fields - Linked Lists do not allow effcient random access to items - Arrays allow better memory locality and cace performance than random pointer jumping

Recursive Objects: Lists and arrays can be though of as recursive objects: - **Lists**: Chopping the first element off a linked list leaves a smaller linked list. The same argument works for strings and since removing characters from string and leaves a string. Lists are recursive objects - **Arrays**: Splitting the first $ k $ elements off an $ n $ element array gives two smaller arrays of size $ k $ and $ n - k $ respectively. Arrays are recursive objects.

The significance of thinking of them as recursive objects is that they lead to simpler list processing and effcient divide-and-conquer algorithms such as quicksort and binary search

2 Stacks and Queues

We use the term *container* to denote a data structure, that permits storage and retrivel of data items *independent of content.* Containers are distinquished by their particular retrieval order they support

Stacks: > Support retrieval by last-in, first-out (LIFO) order. Stacks are simple to implement and very and very efficent. For this reason, stacks are probaly the right container to use when retrieval order dosen't matter at all, such as when processing batch jobs. The put and get operations for stacks are usually called push and pop - Push(x, s): Insert item $ x $ at the top of stack $ s $ - Pop(s): Return (and remove) the top item of stack $ s $

$ LIFO $ order arises when you have to execute recursive algorithms

Queues: > Support retrivel in first in, first out (FIFO) order. The purpose of this order is the minimize the max time spent waiting. The average time will be the same regardless of whether FIFO or LIFO is used. Order is important - Enqueue(x, y): Insert item $ x $ at the back of queue $ q $ - Dequeue(q): Return (and remove) the front item from queue $ q $

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

Operations: - 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): Retrives the item with the largest (or smalelst) 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 is immediately before (or after) $ k $ in sorted order. These enable us to iterate through the elements of the data structure

More powerful dictionary Implementations exist such as binary search trees and hash tables

3.1 Dictionary Sorted vs. Unsorted

- **Search**:
  - Implemented by testing the search key $ k $ against each element of an unsorted array. Thus search takes linear time in the workst case, which is when the key $ k $ is not found in $ A $
- **Insertion**:
  - 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**:

  - We do not need to find the element to delete (because we have index) but we do need to shift once its deleted if its is not in the end. So a clever tick is simply to just write over $ A[x] $ and the decrement $ n $. This only take constant time
- **Pre-/Successor**:
  - refer to the item appearing before/after $ x $ in sorted order. In an unsorted array, the elements physical predecessor is not necessarily its logical predecessor. Both require a sweep through all $ n $ elements of $ A $ to determine the winner
- **Min/Max**:
  - are defined with respect to sorted order, and so require linear sweeps to identiy

If we sort the order, we completely reverse the time complexities. Searches can now be done in $ O(log n) $ time, using binary search because we know the median element. The search continues recursively on the appropiate position. Number of halving of $ n $ untill we get to a single element is $ [lg n]$. Insertion and deletion become expensive because we cant just simply cope the value and reduce size of the array, we must move large portions of the array

3.2 Dictionaries on Lists

- **Insertion/Deletion**:
  - difficult on singly linked list because we need pointer to previous. Thus we must spend linear time searchig for it. Deletion is faster for sorted doubly-linked list than sorted arrays, because splicing out the deleted element from the list is more effcient than filling the hole by moving array elements. The predecessor pointer problem again complicates deletion from singly-linked sorted lists
- **Search**:
  - Sorting provides less benefit for linked lists than it did for arrays. Binary search is no longer possible, becase use can't access the median element without traversing all the elements before it.
- **Traversal Operations**:
  - the predecessor pointer problem again complicates implementing predecessor. In sorted lists, the pre/succ can be implemented in constant time
- **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. We can maintain a separate ponter to the list tail, provided we pay the maintence costs for this pointer on every insertion and deletion. The tail pointer can update in costant time on doubly-linked lists (on insertion check whether last is next or still NULL; on deletion set last to point to the list predecessor of last if last element is deleted). On singly-linked lists, we cannot find predecessor but the the max is on average O (1) because we put the charge on each deletion, which is already linear time

4 Binary Search Trees

Figure 2: The five distinct binary search trees on three nodes

We have had data structures that allow fast search or flexable update but not fast search and flexable update.

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

A rooted binary tree is recursively defined as either \$ (1) \$ empty or \$ (2) \$ consisting of a node called the root, together with two rooted binary trees called the left and right subtrees, respectibely.

A binary search tree labels each node in a binary tree with a single key such that for any node labled \$ x \$, all nodes in the left subtree of \$ x \$ have \$ keys < x \$ while all nodes in the right substree of \$ x \$ have \$ keys > x \$. This labeling is what makes it a BST

4.1 BST Implementation

Binary trees have `left` and `right` pointer feilds and an optional `parent` pointer and a data feild.

Figure 3: Relationships in a binary search tree (left). Finding the minimum and maximum elements in a binary search tree

```
typedef struct tree {
    item_type item; /* data item */
    struct tree *parent; /* pointer to parent */
    struct tree *left; /* pointer to left child */
    struct tree *right; /* pointer to right child */
} tree;
```

4.1.1 Searching in a Tree

BST tree labeling uniquely identites where each key is located. Start at the root, unless its the key you want, proceede either left or right depending upon whether \$ x \$ occurs before or after the root key.

This works due to the recursive nature of the BST structure.

```
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) );
}
```

The search algorithm runs in \$ O (h) \$ time, where \$ h \$ denotes the height of the tree

4.1.2 Finding Min and Max Elements in a Tree

By definition, the smallest key must reside in the left subtree of the root, since all keys in the left subtree have values less than that of the root. The minimum element must be leftmost descent of the root.

```
tree *find_minimum(tree *t)
{
    tree *min; /* pointer to minimum */
    if (t == NULL) return(NULL);

    min = t;
    while (min->left != NULL)
        min = min->left;
    return(min);
    }
```

4.1.3 Traversal in a Tree

Binary search trees make it easy to report the labes in a sorted order. By definition all the keys smaller than the root must lie in the left subtree of the root and all keys bigger than the root in the right subtree. Thus visiting the nodes recursively in accord with such a policy produces an $ in-order $ traversal of the search tree

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

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

4.1.4 Insertion In a Tree

There is only one palce to insert an item $ x $ into a BST $ T $ where we know we can find it again. We must place the $ NULL $ pointer found in $ T $ after an unsuccessful query for the key $ k $

Insert Tree: - a pointer $ l $ to the pointer linking the search subtree to the rest of the tree - the key $ x $ to be inserted - a $ parent $ pointer to the parent node containing $ l $

the node is allocated and linked in on hitting the $ NULL$ pointer.

Note that we pass the pointer to the appropiate left/right pointer in the node during the search, so the assignment $ *l = p $; links the new node into the tree

```
insert_tree(tree **l, item_type x, tree *parent)
{

tree *p; /* temporary pointer */

if (*l == NULL) {
    p = malloc(sizeof(tree)); /* allocate new node */
    p->item = x;
    p->left = p->right = NULL;
    p->parent = parent;
    *l = p; /* link into parent's record */
    return;
}

if (x < (*l)->item)
    insert_tree(&((*l)->left), x, *l);
else
    insert_tree(&((*l)->right), x, *l);
}
```

Allocating the node and linking it in the tree is a constant-time operation after the search has been performed in $ O (h) $ time

4.1.5 Deletion from a Tree

Figure 4: Deleting tree node with $ 0 $, $ 1 $ and $ 2 $ children

Deleting it hard because it meand relinking it descendants subtrees back into the tree somewhere else

If leaf nodes have no children, it may be deleted by simply clearing the pointer to the given node

Having only one chold is straightfordward, there is one parent and one grandchild, and we can link the grandchild directly to the parent without violating the in-order labeling proprty of the tree

The trickey part is when you have two children. One solution is to relabel this node with the key of its immediate succesor in sorted order. The successor must be the smalles key value in the right subtree, specifically the leftmost descendant in the right subtree $ (p) $. Moving this to the point of eletion results in a proprly-labeled binary search tree and reduces our deletion problem to physically removing a node with at least one child

Every deletion requirres the cost of at most two search operations, each talking $ O(h) $ time where $ h $ is the height of the tree, plus a constant amount of pointer manipulation

4.2 How Good Are Binary Search Trees?

When implemented using binary search tree, all three dictionary operations take $ O(h) $ time, where $ h $ is the height of the tree.

The smallest height we can hope for occurs when the tree is perfectly balanced, where $ h = [log n] $. But the tree must be balanced

The insertion algorithms puts each new item at a leaf ndoe where it should have been found. This makes the shape (and more importantly height) of a tree a function of the order in which we insert the key.

Insert is bad because it may produce a skewed tree. Imagine the following operations

$$insert(a)->insert(b)->insert(c)->insert(d)->$$

This would make a skinny linear height tree where only right pointers are used

Most likely our insert time will be $ O (log n) $

4.3 Balanced Search Trees

There are algorithms that can guarantee the height of the tree is always $ O (log n) $. And therefore all dictionary operations (insert, delete, query) take $ O (log n) $ time.

5 Priority Queues

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

Operations: - Insert(Q, x): - Given an item $ x $ with key $ k $, insert it into the priority queue $ Q $ - Find-Max/Min(Q): - Return a pointer to the item whos key value is smaller/larger than any other key in the priority queue $ Q $ - Delete-Min/Max(Q): - Remove the item from the priority Queue $ Q $ whose key is min/max

5.1 Proproty Queue Time Complexity

All priority queue deletions involve only the minimum element. By storing the sorted array in reverse order (largest value on top), the min element will be the last one in the array. Deleting the tail element requires no movment of any items, just decrementing the number of remaining items $ n $, and so delete-minimum can be implemented in constant time

The table above claims that we can implement find-minimum in constant time for each data structure. The trick is using an extra variable to store a pointer/index to the min enetry in each of these structures. We just return the value when asked to find-min.

Finding the new min value if we deleted the old one takes log take linar time in an unsorted array and log time on a tree. This is because we amortize the cost with the delete

A heap is a particular type of a priortiy queue implementation

6 Hashing and Strings

Hash tables are a very pratical way to maintain a dictionary

In python a dictionary implements a hash table underneath it all. The advantage over an array is $ O (1) $ lookup. So if i want to look up a value in an array, i must first loop through the array and when it gets to the value, return it. This can take $ O (n) $ time

Hask tables exploit the fact that an item up in an array takes constant time once you have its index

A hash function is a mathematical function that maps keys to integers. We will use the value of our hash function as an index into an array, and store our item at that position

The first step of the hash function is usually to map each key to a big integer. Let $ a $ be the size of the alphabet on which a given string $ S $ is written. Let $ char(c) $ be a function that mpas each symbol of the alphabet to a unique interger from $ 0 $ to $ a - 1 $

The function above maps each string to a unique (but large) integer by treating the characters of the string as “digets” in a $ base-a $ number system

The result is a unique identifier numbers, but they are so large thay will exceed the number of slots in our hash table $ m $. We must reduce this number to an integer between $ 0 $ and $ m - 1 $ by taking the remainder of $ H(S) $ mod $ m $

If the table size is selected with engough finesse ($ m $ is a large prime not close to $ 2^i - 1 $, the resulting hash values should be fairly uniformly distributed

6.1 Collision Resolution

6.1.1 Chaining

Two distinct keys may hash to the same value. *Chaining* is the easiest approach to collision resolution

Figure 5: Collision resolution by chaining

Represent the hash table as an array of $ m $ linked lists as shown above

The $ ith $ list will contain all the items that hash to the value of $ i $. Thus search, insertion and deletion reduce to the corresponding problem in linked lists

If the $ n $ keys are distributed uniformly in a table, each list will contain roughly $ n/m $ elements, making them a constant size when $ m = n $

Chaining is very natural, but devotes a considerbale amount of memory to pointers. This is space that could be used to make the table larger, and hence the “lists” smalelr

6.1.2 Open Adressing

Figure 6: Collision resolution by open adressing

An Alternative to chaining is open adressing. The hash table is maintained as an array of elements (not buckets), each initialized to $ NULL $, as shown above

On insertion, we check to see if the desired position is empty or not

If empty, we insert. Else we find another place for insert. The simplest technique called *sequential probing* inserts the item in the next open spot in the table

If the table is not too full, the contiguous runs of items should be fairly small, hence this location should be only a few slots from its intended position

Searching for a given key now involves going to the appropriate hash vaule and checking to see if the item there is the one we want. If its there, return it or keep checking through the lenght of the run

Deletion in an open adressing 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

Chaining and open adressing both require $ O (m) $ to initialize an *m*-element hash table to null elements prior to the first insertion

Traversing all the elements in the table takes $ O(n + m) $ times for chaining, because we have to scan all $ m $ buckets looking for elements, even if the actual number of inserted items is small. This reduces to $ O (m) $ time for open adressing since $ n $ must be at most $ m $

When using chaining with doubly-linked lists to resolve collisions in an $ m$-element hash table, the dictionary operations for $ n $ items can be implemented in the following expected and worst case times

Pragmatically, a hash table is often the best data structure to maintain a dictionary

6.2 Efficent String Matching via Hashing

Strings are sequences of characters where the order of the characters matter.

The primary data structure for representing strings in an array of characters. This allows for constant-time access to the $ ith $ character of the string.

Generally for substring matching, the time complexity can be $ O (nm) but we can expect linear time using Rabin-Karp algorithm, which is based on hashing

Suppose we compute a given hash function on both the pattern sting $ p $ and the $ m$-charcter substring starting from the $ ith $ position. If the strings are identical, the resulting hash values must be the same, else they are different.

We can easily spend the $ O(m) $ time it takes to explicitly check the identity of two strings whenever the hash value agree

The catch is that it takes $ O (m) $ time to compute a hash function and thus we are left with $ O(mn) $ algorithm again

Lets apply the previously defined hash function to the $ m $ characters starting from the $ jth $ position of string $ S $

What changes if we now try to compute $ H(S, j + 1)$? The hash of the next window of $ m $ characters?

A little algebra reveals that

This means that once we know the hash value form the $ j $ position, we can find the hash value from the (j + 1)st position for the cost of two multiplications, one addition and one subtraction

This can be done in constant time (the value of $a^m - 1$ can be computed once and used for all hash value computations

Rabin-Karp is a good example of a randomized algorithm (if we pick $ M $ in some random way; $ M $ is a reasonably large prime number

There is no guarantee that the algorithm runs in $ O(n + m) time, because we may get unlucky and have the hash values regularly collide with spurious matches. But the odds are so great that there will be no collision

6.3 Duplicate Detection via Hashing

The key idea of hashing is to represent a large object (be it a key, string or a substring) using a single number. The goal is a representation of the large object by an entity that can be manipulated in constant time, such tht it is relatively unlikely that two different large objects map to the same value

Features of Hashing: - Tells you if a document is different from all the rest in a large corpus? - Tells us if the document is plagiarized rom a document in a large corpus? - Uses cryptographic hashing to make sure no one reads the bids

7 Specialized Data Structure

- **String Data Structures**:
  - Character stings are typically represented by arrays of characters, sometime with a special character to mark the end of a string. Suffix trees/arrays are special data structures that preprocess strings to make pattern matching operations faster
- **Geometric Data Structures**:
  - Geometric data types typically consists of collections of data points and regions. Regions in the place can be described by polygons where the boundary of the polygon is given by a chain of line semgents. Polygons can be represented using an array of points \$ (v_1, ..., v_n, v_1) \$, such that \$ (v_i, v_i+1) \$ is a segment of the boundary. Spatial data structures such as \$ kd-trees \$ organize points and regions by geometric locations to support fast search
- **Graph Data Structures**:
  - Graphs are typically represented using either adjaceny matrices or adjacency lists. The choice of representation can have a substantial impact on the design of the resulting graph algortihm
- **Set Data Structure**:
  - Subset of items are typically represented using a dictionary to support fast membership queries. Alternately, bit vectors are boolean arrays such that the \$ ith \$ bit represents true if \$ i \$ is in the subset.