[Data Structures Overviews]

Design and Analysis of Algorithms (CSc 523)

Samujjwal Bhandari
Central Department of Computer Science and Information Technology (CDCSIT)
Tribhuvan University, Kirtipur,
Kathmandu, Nepal.

Since the main aim of this part is to introduce some of the data structures if you want rigorous study you can always consult the book on Data Structures.

Simple Data structures

The basic structure to represent unit value types are bits, integers, floating numbers, etc. The collection of values of basic types can be represented by arrays, structure, etc. The access of the values are done in constant time for these kind of data structured.

Linear Data Structures

Linear data structures are widely used data structures we quickly go through the following linear data structures.

Lists
Stacks and queues

a. Lists

List is the simplest general-purpose data structure. They are of different variety. Most fundamental representation of a list is through an array representation. The other representation includes linked list. There are also variety of representations for lists as linked list like singly linked, doubly linked, circular, etc.
There is a mechanism to point to the first element. For this some pointer is used. To traverse there is a mechanism of pointing the next (also previous in doubly linked). Lists require linear space to collect and store the elements where linearity is proportional to the number of items. For e.g. to store n items in an array nd space is require were d is size of data. Singly linked list takes n(d + p), where p is size of pointer. Similarly for doubly linked list space requirement is n(d + 2p).

Array representation(ordered list)

- Operations require simple implementations.
- Insert, delete, and search, require linear time, search can take O(logn)!!
- Inefficient use of space

Singly linked representation (unordered)

- Insert and delete in O(1) time, for deletion pointer must be given otherwise O(n).
- Search in O(n) time
- Memory overhead, but allocated only to entries that are present.

Doubly linked representation

- Insert and delete in O(1) time !!

Operations on lists

create(): create an empty list
destroy():
IsEmpty()
Length():
Find(k): find k'th element
Search(x): find position of x
delete():
insert(x): insert x after the current element element
and plenty of others

b. Stacks and Queues

These types of data structures are special cases of lists. Stack also called LIFO (Last In First Out) list. In this structure items can be added or removed from only one end. Stacks are generally represented either in array or in singly linked list and in both cases insertion/deletion time is O(1).

Operations on stacks

Create()	IsEmpty()	IsFull()
Top()	push(x)	pop()

The queues are also like stacks but they implement FIFO(First In First Out) policy. One end is for insertion and other is for deletion. They are represented mostly circularly in array for O(1) insertion/deletion time. Circular singly linked representation takes O(1) insertion time and O(n) deletion time, or vice versa. Representing queues in doubly linked list have O(1) insertion and deletion time.

Operations on queues

Create()	IsEmpty()	IsFull()
First()	Last()	Enqueue(x)
Dequeue()		

Tree Data Structures

Tree is a collection of nodes. If the collection is empty the tree is empty otherwise it contains a distinct node called root (r) and zero or more subtrees whose roots are directly connected to the node r by edges. The root of each tree is called child of r, and r the parent. Any node without a child is called leaf (you may find other definition from any data structure book). We can also call the tree as a connected graph without a cycle. So there is a path from one node to any other nodes in the tree. The main concern with this data structure is due to the running time of most of the operation require O(logn).we can represent tree as an array or linked list.

Some of the definitions

- Level *h* of a full tree has d^{h-1} nodes.
- The first *h* levels of a full tree have

  $$1 + d + d^2 + \cdots + d^{h-1} = \frac{d^h - 1}{d - 1}$$

  nodes.
- A tree of height *h* and degree *d* has at most d^h - 1 elements

Remember the following terminology tree, subtree, complete tree, root, leaf, parent, children, level, degree of node, degree of tree, height, tree traversal, etc.

Priority Queue Trees

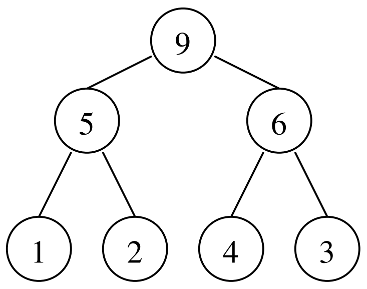
Heap

A heap is a complete tree with an ordering-relation R holding between each node and its descendant. *Note that the complete tree here means tree can miss only rightmost part of the bottom level.*

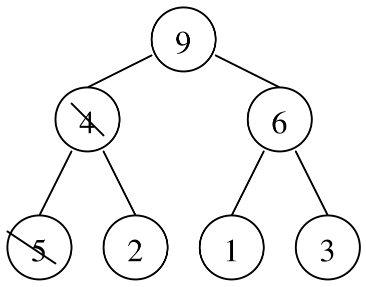
R can be smaller-than, bigger-than.

E.g. Heap with degree 2 and R is "bigger than".

Heap



Not heap



Applications

Priority Queue A dynamic set in which elements are deleted according to a given ordering-relation.

Heap Sort Build a heap from the given set ($O(n)$) time, then repeatedly remove the elements from the heap ($O(n \log n)$).

Implementation

An array

Insertion and deletion of an element takes O(*log* n*) time. More on this later.*

Search Trees

a. Binary Search Trees

BST has at most two children for each parent. In BST a key at each vertex must be greater than all the keys held by its left descendents and smaller or equal than all the keys held by its right descendents.

Searching and insertion both takes O(h) worst case time, where h is height of tree and the relation between height and number of nodes n is given by log n < h+1 <= n. for e.g. height of binary tree with 16 nodes may be anywhere between 4 and 15.

When height is 4 and when height is 15?

So if we are sure that the tree is height balanced then we can say that search and insertion has $\Theta(\log n)$ run time. Other wise we have to content with $\Theta(n)$.

b. AVL Trees

Balanced tree named after Adelson, Velskii and Landis. AVL trees consist of a special case in which the subtrees of each node differ by at most 1 in their height (Read more on your own like rotations, insertions, deletions, etc.).

c. Red Black Trees

A binary search tree in which

- All nodes are either red or black.
- The root and leaves are colored black.
- All the paths from the root to the leaves agree on the number of black nodes
- No path from the root to a leaf may contain two consecutive nodes colored red
- Red nodes have only black children

Empty subtrees of a node are treated as subtrees with roots of black color.
The relation $n \geq 2^{h/2} - 1$ implies the bound $h \leq 2 \log_2(n + 1)$.

Insertion

First insert new node as in BST. If it is root color it black else color it red. When we color the non-root node as red then many cases arises some of them violating the red black tree properties.

Case 1: if the parent is black, it is ok

Case 2: If the parent and parent's sibling (better say uncle) are red. Change the color of parent, uncle and grandparent. Some node g (parent, uncle and grandparent) may be root if so do not change the color. This process will introduce property violation in upper part so continue the process.

Case 3: Red parent, Black uncle; the node is "outside". Rotate the parent about grandparent and recolor them.

Case 4: Red parent, Black uncle; the node is "inside". Double rotation; recolor node and grandparent.

*Note: normal insertion running time O(h) i.e. O(log n) [*h $\leq$ *2 log* $_2$*(*n + *1)] and the rotation needs extra O(1)time.*

c. Multiway Trees

A m-way search tree is a tree in which

- The nodes hold between 1 to m-1 distinct keys
- The keys in each node are sorted
- A node with k values has k+1 subtrees, where the subtrees may be empty.
- The i^th^ subtree of a node $[v_1, ..., v_k]$, $0 < i < k$, may hold only values v in the range $v_i < v < v_{i+1}$ (v_0 is assumed to equal $-\infty$, and v_{k+1} is assumed to equal *infinity*).

A m-way tree of height h has between h and $m^h - 1$ keys.

Insertion

- Search the key going down the tree until reaching an empty subtree
- Insert the key to the parent of the empty subtree, if there is room in the node.
- Insert the key to the subtree, if there is no room in its parent.

Deletion

- If the key to be deleted is between two empty subtrees, delete it
- If the key is adjacent to a nonempty subtree, replace it with the largest key from the left subtree or the smallest key from the right subtree.

d. B Trees

A m-way tree in which

- The root has at least one key
- Non-root nodes have at least $\lceil m/2 \rceil$ subtrees (i.e., at least $\lfloor (m - 1)/2 \rfloor$ keys)
- All the empty subtrees (i.e., external nodes) are at the same level

B-trees are especially useful for trees stored on disks, since their height, and hence also the number of disk accesses, can be kept small.

The growth and contraction of m-way search trees occur at the leaves. On the other hand, B-trees grow and contract at the root.

Insertion

- Insert the key to a leaf
- Overfilled nodes should send the middle key to their parent, and split into two at the location of the submitted key.

Deletion

- Key that is to be removed from a node with non-empty subtrees is being replaced with the largest key of the left subtree or the smallest key in the right subtree. (The replacement is guaranteed to come from a leaf.)
- If a node becomes under staffed, it looks for a sibling with an extra key. If such a sibling exists, the node takes a key from the parent, and the parent gets the extra key from the sibling.
- If a node becomes under staffed, and it can't receive a key from a sibling, the node is merged with a sibling and a key from the parent is moved down to the node.

e. Splay Trees

These trees do not keep extra balancing data so also known as self-adjusting trees. Splaying a node means moving it up to the root by the help of rotation. Most of the rotation are double and possibly single at the ends. Double rotations have two cases “zig-zig” and “zig-zag”.

Here let x is the non-root node on the access path at which we are rotating. If the parent of x is the root we just rotate x and the root otherwise x has both parent and grandparent, here we need to consider above-mentioned two cases. In the case of “zig-zag” x is a right child and p is the left child (or vice versa). Otherwise we will have “zig-zig” case where both x and p lies on the same side i.e. they are both, either left children, or right children.

Operations

Search: Find the node with a given key if not found splay key’s predecessor or successor node.

Insertion: Insert new node as in other case and splay it.

Deletion: Find the node to be deleted, splay it and delete.

Complexity

Splay trees are not explicitly balanced but with each new operation it tends to be more balanced. The main idea behind splay tree is that even though some operation takes O(n) times the but for m operation it takes O(mlogn) time where O(logn) is amortized running time for each operation. Another concept is 90-10 rule i.e. in practice 90% of processes access 10% of data.

Suggested Exercises

1. The *level order* listing of the node of a tree first lists the root, then all the nodes of depth 1, then all the nodes of depth 2, and so on. Order of listing of nodes of the same level is left to right. Write an algorithm to list the nodes of a tree in *level order (writing program also is good)*.
2. Give an algorithm to sort 5 elements with 7 comparisons.