Chapter 2

Review of 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

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 varieties 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 required 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

- ✓ Operations require simple implementations.
- ✓ Insert, delete, and search, require linear time, search can take O(logn) if binary search is used. To use the binary search array must be sorted.
- ✓ Inefficient use of space

Singly linked representation (unordered)

- ✓ Insert and delete can be done in O(1) time if the pointer to the node is given, otherwise O(n) time.
- ✓ Search and traversing can be done in O(n) time
- ✓ Memory overhead, but allocated only to entries that are present.

Doubly linked representation

- ✓ Insert and delete can be done in O(1) time if the pointer to the node is given, otherwise O(n) time.
- ✓ Search and traversing can be done in O(n) time
- ✓ Memory overhead, but allocated only to entries that are present, search becomes easy.

Some Operation with List

- boolean isEmpty ();

  Return true if and only if this list is empty.
- int size ();

  Return this list's length.
- boolean get (int i);

  Return the element with index i in this list.
- boolean equals (List a, List b);

  Return true if and only if two list have the same length, and each element of the lists are equal
- void clear ();

  Make this list empty.
- void set (int i, int elem);

  Replace by elem the element at index i in this list.
- void add (int i, int elem);

  Add elem as the element with index i in this list.
- void add (int elem);

  Add elem after the last element of this list.
- void addAll (List a List b);

  Add all the elements of list b after the last element of list a.
- int remove (int i);

  Remove and return the element with index i in this list.
- void visit (List a);

  Prints all elements of the list

Operation	Array representation	SLL representation
get	$O(1)$	$O(n)$
set	$O(1)$	$O(n)$
add(int,data)	$O(n)$	$O(n)$
add(data)	$O(1)$	$O(1)$
remove	$O(n)$	$O(n)$
equals	$O(n^2)$	$O(n^2)$
addAll	$O(n^2)$	$O(n^2)$

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), but search time is O(n).

Operations on stacks

- **boolean** isEmpty ();

  Return true if and only if this stack is empty. Complexity is O(1).

- int getLast ();

  Return the element at the top of this stack. Complexity is O(1).

- **void** clear ();

  Make this stack empty. Complexity is O(1).

- **void** push (int elem);

  Add elem as the top element of this stack. Complexity is O(1).

- int pop ();

  Remove and return the element at the top of this stack. Complexity is O(1).

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(1) deletion time. Again Representing queues in doubly linked list have O(1) insertion and deletion time.

Operations on queues

- boolean isEmpty ();

  Return true if and only if this queue is empty. Complexity is O(1).

- int size ();

  Return this queue's length. Complexity is O(n).

- int getFirst ();

  Return the element at the front of this queue. Complexity is O(1).

- void clear ();

  Make this queue empty. Complexity is O(1).

- void insert (int elem);

  Add elem as the rear element of this queue. Complexity is O(1).

- int delete ();

  Remove and return the front element of this queue. Complexity is O(1).

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 sub-trees 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. 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

  $1 + d + d^2 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots d^{h-1} = (d^h - 1)/(d-1)$

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 O(log n) run time otherwise we have to content with O(n).

Operation	**Algorithm**	**Time complexity**
Search	BST search	*O*(log *n*) best *O*(*n*) worst
Add	BST insertion	*O*(log *n*) best *O*(*n*) worst
Remove	BST deletion	*O*(log *n*) best *O*(*n*) worst

AVL Trees

Balanced tree named after Adelson, Velskii and Landis. AVL trees consist of a special case in which the sub-trees of each node differ by at most 1 in their height. Due to insertion and deletion tree may become unbalanced, so rebalancing must be done by using left rotation, right rotation or double rotation.

Operation	**Algorithm**	**Time complexity**
Search	AVL search	*O*(log *n*) best, worst
Add	AVL insertion	*O*(log *n*) best, worst
Remove	AVL deletion	*O*(log *n*) best, worst

Priority Queues

Priority queue is a queue in which the elements are prioritized. The least element in the priority queue is always removed first. Priority queues are used in many computing applications. For example, many operating systems used a scheduling algorithm where the next process executed is the one with the shortest execution time or the highest priority. Priority queues can be implemented by using arrays, linked list or special kind of tree (I.e. heap).

- boolean isEmpty ();

  Return true if and only if this priority queue is empty.

- int size ();

  Return the length of this priority queue.

- int getLeast ();

  Return the least element of this priority queue. If there are several least elements, return any of them.

- void clear ();

  Make this priority queue empty.

- void add (int elem);

  Add elem to this priority queue.

- int delete();

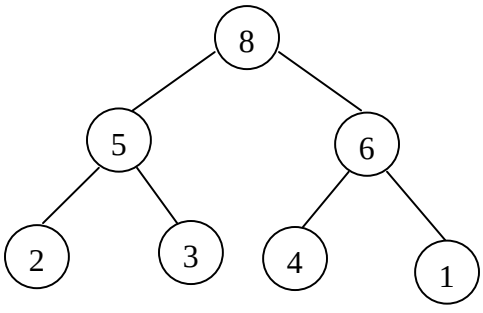
  Remove and return the least element from this priority queue. (If there are several least elements, remove the same element that would be returned by getLeast.

Operation	**Sorted SLL**	**Unsorted SLL**	**Sorted Array**	**Unsorted Array**
add	$O(n)$	$O(1)$	$O(n)$	$O(1)$
removeLeast	$O(1)$	$O(n)$	$O(1)$	$O(n)$
getLeast	$O(1)$	$O(n)$	$O(1)$	$O(n)$

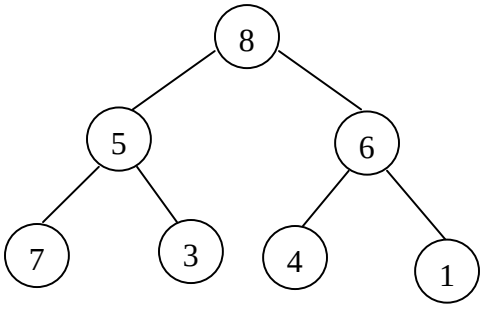
Heap

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 a heap

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

Heaps are implemented by using arrays. Insertion and deletion of an element takes O(log n) time. More on this later

Operation	**Algorithm**	**Time complexity**
add	insertion	$O(\log n)$
delete	deletion	$O(\log n)$
getLeast	access root element	$O(1)$

Priority Queues

Priority queue is a queue in which the elements are prioritized. The least element in the priority queue is always removed first. Priority queues are used in many computing applications. For example, many operating systems used a scheduling algorithm where the next process executed is the one with the shortest execution time or the highest priority. Priority queues can be implemented by using arrays, linked list or special kind of tree (I.e. heap).

- boolean isEmpty ();

  Return true if and only if this priority queue is empty.

- int size ();

  Return the length of this priority queue.

- int getLeast ();

  Return the least element of this priority queue. If there are several least elements, return any of them.

- void clear ();

  Make this priority queue empty.

- void add (int elem);

  Add elem to this priority queue.

- int delete();

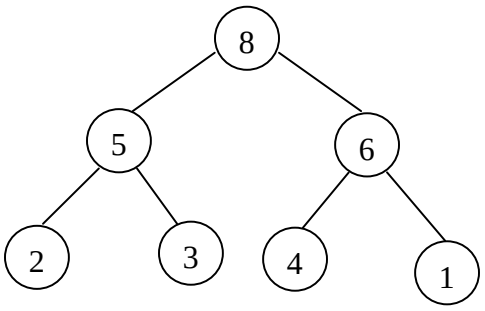
  Remove and return the least element from this priority queue. (If there are several least elements, remove the same element that would be returned by getLeast.

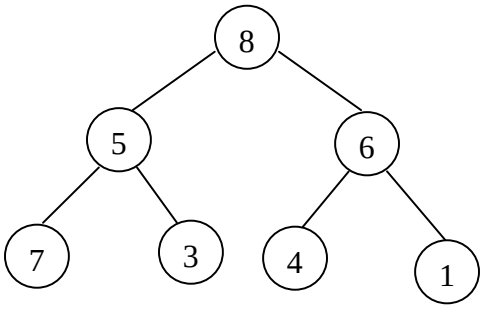
Operation	**Sorted SLL**	**Unsorted SLL**	**Sorted Array**	**Unsorted Array**
add	$O(n)$	$O(1)$	$O(n)$	$O(1)$
removeLeast	$O(1)$	$O(n)$	$O(1)$	$O(n)$
getLeast	$O(1)$	$O(n)$	$O(1)$	$O(n)$

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

Implementation

Heaps are implemented by using arrays. Insertion and deletion of an element takes O(log n) time. More on this later

Operation	**Algorithm**	**Time complexity**
add	insertion	$O(\log n)$
delete	deletion	$O(\log n)$
getLeast	access root element	$O(1)$

Hash Table

A hash table or hash map is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person's name), to their associated values (e.g., their telephone number). The hash function is used to transform the key into the index of an array element where the corresponding value is to be sought. Ideally, the hash function should map each possible key to a unique slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created). Instead, most hash table designs assume that *hash collisions*—different keys that map to the same hash value—will occur and must be accommodated in some way.

In a well-dimensioned hash table, the average cost each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at constant average (i.e O(1)). In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.