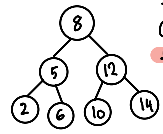
# Symbol Table API

| | Ordered array | | | Unordered LL | | | BST | | | RB | 2 - 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Search | Insert | delete | | | | | | | | |
| Worst | lg N | N (Shifting) | N | N | O(1) | N | N | N | N | ideal → logN | log N |
| AVG | lg N | N/2 | N/2 | N/2 | N | N/2 | lgN | lgN | ? | logN | log N |

rank: number of keys less than the given key.

floor: Highest node less than or equal to key.

Ceiling: Smallest node greater than or equal to the key.



floor (9) = 8

Ceil (9) = 10

floor or ceiling of a key in the tree is itself.

## LL — Symbol Table Methods by implementation

```
get(Key akey){
  for(Node x = first; x! = null; x=x.next){
    if(akey.equals){
      return x.val;
    }
  return null;
}
}
```

Put (Pseudo)

```
Put(key, val){
if key = null{
  delete(key);
  return;
}
for(){
  Check dupes + update
  return;
}
first = new Node()
}
```

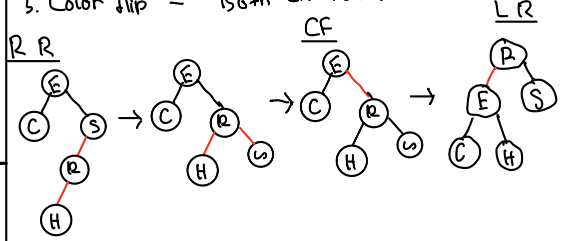
delete

```
delete(key){
first = delete(first, key)
N--;
}
delete(x, key){
if(x == null)
  return x.next;
x.next = delete(x.next, key);
return x;
}
```

Recursive rank (4 cases)

1. Key is null return 0.
2. Key < target (recurse left)
3. Key > target (1 + Size(x.left) + rank(x.right))
4. key == target(recurse left).

1. Left rotate - Right child red, left child black.
2. Rotate right - Left child + left GC is red.
3. Color flip - Both children red




RR occurs with 2 consec. left leaning red links.

LR occurs when right is red and left is black or null.

CF occurs when Left link and Right link are both Red.

## OA

```
get(key){
  r = rank(key);
  if(CheckFor(key, r)){
    return Val[r];
  }
  return null;
}
```

Put

```
void put(key, val){
  if(N >= key.length){
    resize(key.length * 2);
  }
  r = rank(key)
  if(!checkFor(key, rank)){
    ShiftRight(r);
    keys[r] = key;
    N++;
  }
  vals[r] = val;
}
```

delete

```
void delete(key){
  int r = rank(key);
  if(checkFor(key, r)){
    ShiftLeft(r);
    N--;
  }
}
ShiftLeft(int loc){
  for(int i = loc, i < N-1; i++){
    keys[i] = key[i+1];
    vals[i] = val[i+1];
  }
  keys[N-1] = null;
  vals[N-1] = null;
}
```

CHECK ALL Functions that Mutate for Size

LL Floor

```
key floor(key){
  if(key == null)
    return null;
  }
  key best = null;
  for(Node current = first; current != null; current = current.next){
    int cmp = key.compareTo(current.key);
    if(cmp == 0){
      return current.key;
    }
    if(cmp > 0){
      if(best == null || current.key.compareTo(best) > 0){
        best = current.key;
      }
    }
  }
  return best;
}
```

## BST

```
K get(){
  return get(root, key);
}
get(x, key){
if(x == null){
  return null;
}
int cmp = key.compareTo(x.key);
if(cmp > 0){
  return get(x.right, key);
}
else if(cmp < 0){
  return get(x.left, key);
} else{
  return x.val;
}
}
```

Put

```
Put(key, val)
  root = Put(root, key, val);
}
Put(x, key, val){
  if(x == null){
    return new Node(key, val, 1)
  }
  int cmp = key.compareTo(x.key);
  if(cmp > 0){
    x.right = Put(x.right, key, val);
  }else if(cmp < 0){
    x.left = Put(x.left, key, val);
  }else{
    x.val = val;
  }
  x.size = 1 + size(x.left) + size(x.right);
  return x;
}
```

delmin

```
deleteMin(){
  root = deleteMin(root);
}
deleteMin(x){
  if(x.left == null){
    return x.right;
  }
  x.left = deleteMin(x.left);
  x.size = 1 + size(x.left) + size(x.right);
  return x;
}
```

Binary Search Rank

```
int counter = 0;
int lo = 0;
int hi = N-1;
int mid;
while(lo <= hi){
  mid = (hi + lo)/2
  if(key < a[mid]){
    hi = mid - 1;
  } else if(key > a[mid]){
    Counter += (mid - lo) + 1;
    lo = mid + 1;
  } else if(key == a[mid]){
    return mid;
  }
  return Counter
}
```

- Inorder traversal prints keys in ascending order.
- Unique put only changes LL implementation (becomes O(1)).

depth of a node is number of edges from root to node.

height is number of edges below until a null link.

depth of root = 0;

```
Private int sum(Node x){
  if(x == null){
    return 0;
  }
  int sum = x.val
  sum += sum(x.left) + sum(x.right);

  return sum
}
```

* add if statements for specific cases and init significant var to 0, only update it if the if-statement fires

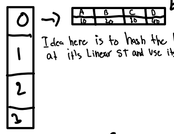
## Standard hashing recipe for user defined ObJs

```
hashCode (){
    int hash = 17;   ← non-zero constant
    hash = 31 * hash + obj.hashcode();
      ↓
    return hash
}
```

## Types of Hashing

1. Open Adressing (Linear Probing)
2. Seperate chaining

SC




Idea here is to hash the Key, find the corresponding index and look at it's Linear ST and use its methods to add or get.

## Linear Probing (optimal load factor is 50%)

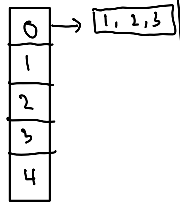
```
Class LinearProbing<Key, Value>{
  Private int M = 100; // size
  Private Value[] vals = (Value[]) new Object[M];
  Private Key [] Keys = (Key[]) new Object[M];
  Private int hashKey(Key key){ return ... % M};

  Public void put(Key key, Value val){
      int i;                                      linear Probing
                                                        ↓
      for( i = hashKey(Key); keys[i] != null; i = (i + 1) % M{
        if( Keys[i].equals(Key) {
            Vals[i] = val;
            return
        }
      }
      Keys[i] = Key;
      Vals[i] = Val;
      N++;
  }
  Public Value get(Key key){
     for(int i = hash(Key); Keys[i] != null; i = (i + 1) % M){
        if (key.equals(Keys[i]) return Vals[i];
        return null;
  }

}
}
```

```
Public boolean equals(Object x){
  if (x == null) return false;
  if (x == this) return true;
  if(x.getClass() != this.getClass()) return false;
  Transaction that = (Transaction) x;
  if (!that.who.equals(this.who)) return false;
  if (!that.when.equals(this.when)) return false;
  if(that.amount != this.amount) return false;
  return true;
}
```

| | Worst | | | Avg | | |
|---|---|---|---|---|---|---|
| seperate chaining | N | N | N | 3-5 | 3-5 | 3-5 |
| linear Probing | N | N | N | 3-5 | 3-5 | 3-5 |

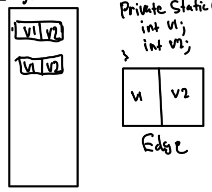
## 3 Graph implementations

### Adjacency List




A list of some collection (Bag) where each index represents a vertex and its stored collection stores adjacent vertices.

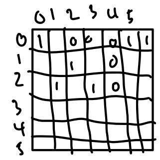
### Edge List




```
Private Static Class Edge {
    int V1;
    int V2;
}
```

A Collection of Edge objects where each edge represents a connection b/w 2 vertices V1 and V2.

### Adjacency Matrix




A matrix that represents a connection b/w other vertices with 1's and 0's

### Real life Applications of DS

Symbol Table: DNS Lookup
Graph: Model friend groups in apps/schedules
DFS – Maze Traversal
BFS – GPS Shortest Distances
Cycle Detection; Schedules

| Implementation | Space | add Edge | Check Adjacency of V, W | iterate through vertice adj to V |
|---|---|---|---|---|
| Edge List | E | 1 | E | E |
| ADJ Matrix | $V^2$ | 1 | 1 | V |
| ADJ Lists | E+V | 1 | degree(V) | degree(V) |

```
Class EdgeList Implementation{
    int V;
    private LinkedList<Edge> edgelist;
    Private class Edge {
      Public int V1, V2;
      Public Edge(V1, V2) {
         this.V1 = V1;
         this.V2 = V2;
      }
    }
    Public Edgelist(int V){
       this.V = V;
       edgelist = new LinkedList<Edge>();
    }
  Public boolean addEdge(V1, V2){
      if(! edgeExists(V1, V2) {
         edgelist.add(new Edge(V1, V2));
         return true;
      }
      return false;
  }
}
```

## Study Guide 9.12 outline

**Connected Graph?**

1. Do 1 DFS pass on a vertex.
2. After the DFS check the marked[] if all are true the it is connected.

**Graph Has Cycle?**

1. Use BFS
2. In BFS loop add else if(marked[W] && edgeTo[V] != W)

```
Private void bfs(Graph G, int s) {
    q = new queue
    marked[S] = true;
    q.enque(s);
    while(! q.isEmpty() {
      int v = q.deque;
      for(int W : G.adj(V) {
        if (! marked[W] {
          edgeTo[W] = V;
          marked[W] = true;
          q.enque(W);
        } else if( marked[W] && edgeTo[V] != W) {
          hasCycle = true
        }
      }
    }
}
```

## Generic BFS + DFS (Use Stack instead)

```
bfs(Graph G, int s) {
    q = new queue;
    marked[S] = true;     // additional arrays (distance, edgeTo)
    q.enque(s);
  while(! q.isEmpty() {
    int v = q.deque();
    for(int W : G.adj(v) {
      if( ! marked[W]) {
        marked[W] = true
        q.enque(W);
      Any additional processing here (distance, edgeTo)
    }
  }
}
```

## Generic Recursive DFS

```
Public void dfs(Graph G, int v) {
    marked[v] = true;
    for (int W : G.adj(V){
       // processing here
       dfs(G, W);
    }
}
```

## Steps to find Strongly Connected Components.

1. Do a DFS (Postorder) adding Nodes to stack. Yields topological order.
2. Reverse Graph, do regular DFS using the topological order, all reachable vertices are connected.