CS61B Week9

Lec 23 Hashing

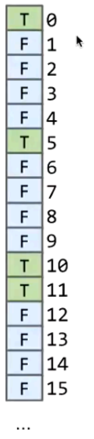
Part 1.Set Impletations,DataIndexedIntegerSet

How to improve the performance of the unordered array in searching?

Using data as an Index

One extreme approach: All data is really just bits.

- Use data itself as an array index.
- Store true and false in the array.




```java
DataIndexedIntegerSet diis = new DataIndexedIntegerSet();
diis.insert(0);
diis.insert(5);
diis.insert(10);
diis.insert(11);
```

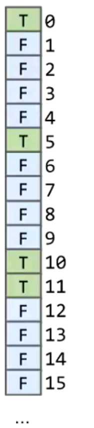
Set containing 0, 5, 10, 11

- Advantages:

  `Insert` and `Contain` takes constant time.
- Disadvantages:A huge waste of memory.We spoil the space to save time.

DataIndexedIntegerSet Implementation

```java
public class DataIndexedIntegerSet {
    boolean[] present;

    public DataIndexedIntegerSet() {
        present = new boolean[16];
    }

    public insert(int i) {
        present[i] = true;
    }

    public contains(int i) {
        return present[i];
    }
}
```



Set containing 0, 5, 10, 11

	contains(x)	insert(x)
Linked List	Θ(N)	Θ(N)
Bushy BSTs	Θ(log N)	Θ(log N)
Unordered Array	Θ(N)	Θ(N)
DataIndexedArray	Θ(1)	Θ(1)

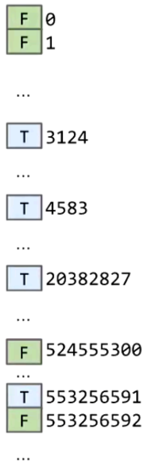
Worst case runtimes

Part 2.Binary Representations,DataIndexedSet

We can think the string into a Base 27 number or Base 32 number(Since there are 26 letters and 32 is 2^5 so is easier to converted into a Base 2 number)

DataIndexedWordSet Implementation

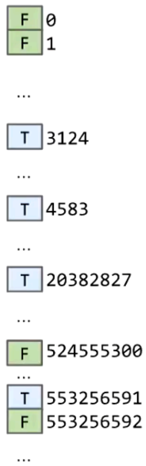
```java
/** Converts ith character of String to a letter number.
  * e.g. 'a' -> 1, 'b' -> 2, 'z' -> 26 */
public static int letterNum(String s, int i) {
    int ithChar = s.charAt(i);
    if ((ithChar < 'a') || (ithChar > 'z'))
        { throw new IllegalArgumentException();
    return ithChar - 'a' + 1;
}

public static int convertToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i++) {
        intRep = intRep << 5; // same as intRep * 32;
        intRep = intRep + letterNum(s, i);
    }
    return intRep;
}
```



DataIndexedArray

Two fundamental challenges:

- How do we resolve ambiguity (“grosspie” vs. “bosspie”)?
    - We’ll call this ***collision handling***.
- How do we convert arbitrary data to an index?
    - We’ll call this ***computing a hashCode***.
    - For Strings, this was relatively straightforward (treat as a base 27 or base 32 number).
    - Note: Java requires that EVERY object provide a method that converts itself into an integer: hashCode()
    - More on what makes a good hashCode() later.

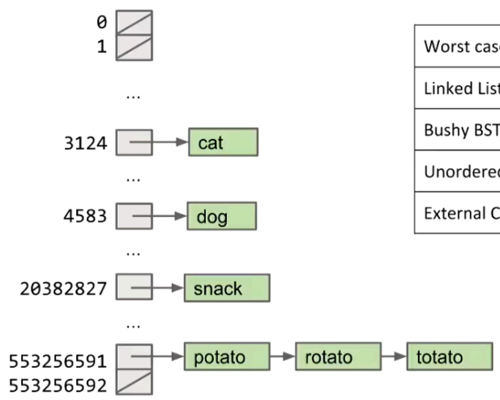


Part 3.Handling Collisions

Suppose N items have the same hashcode h:

Instead of storing true in position h,store a list of these N items at position h

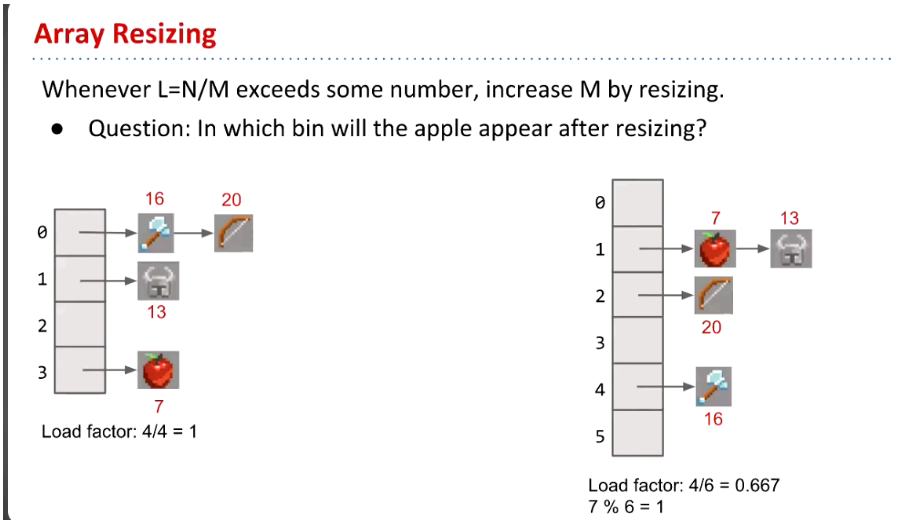
External Chaining: Storing all items that map to h in a linked list.



Worst case time	contains(x)	insert(x)
Linked List	Θ(N)	Θ(N)
Bushy BSTs	Θ(log N)	Θ(log N)
Unordered Array	Θ(N)	Θ(N)
External Chaining	Θ(Q)	Θ(Q)

Q: Length of longest list

If N items are distributed across M bucket,average time grows with N/M=L,also known as the **load factor**.Average runtime is$\Theta(L)$.

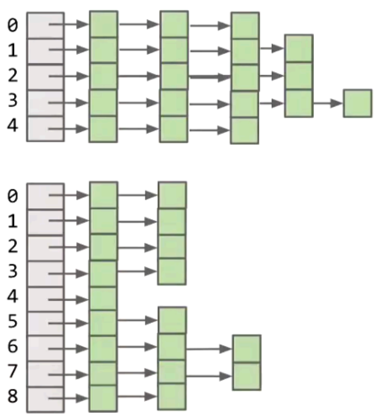


Part 4.Hash Table

This data structure we 've designed is called a **hash table**

1. defination:
    - Every item is mapped to a bucket number using a hash function
    - Typically,computing hash funciton consists of two steps:
        a. Computing a hashCode(Integer between $-2^{31}$and$2^{31}-1$since the largest size of array in java is limited)
        b. HashCode Module M=Computing index
        c. If $L=\frac{N}{M}$ gets too large,increase M.
2. **Solve collisions** of multiple items in the same bucket:
    - Way1:External Chaining:Create a list for each bucket
    - Way2:Open Addressing(a little stranger,not necessarily better,see extra slides)
3. Performance

Assuming items are spread out (e.g. not all in the same bucket):



Average case time	contains(x)	insert(x)
External Chaining, Fixed Size	Θ(L)	Θ(L)
External Chaining With Resizing	Θ(L)	Θ(L)
Balanced BST	Θ(log N)	Θ(log N)

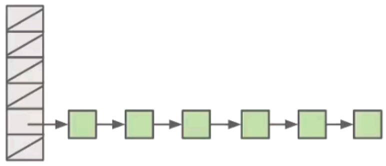
	Load Factor L
External Chaining, Fixed Size	Θ(N)
External Chaining With Resizing	Θ(1)

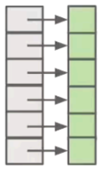
Part 5.Hash Functions

We want elements to be allocated evenly in every bucket rather than gather in the same one.

Goal: We want hash tables that look like the table on the right.

- Want a hashCode that spreads things out nicely on real data.
  - Example #1: return 0 is a bad hashCode function.
  - Example #2: Our convertToInt function for strings was bad. Top bits were ignored, e.g. "potato" and "give me a potato" have same hashCode.
- Writing a good hashCode() method can be tricky.





Our convertToInt function:

- $h(s) = (s_0 - 'a' + 1) \times 32^{n-1} + (s_1 - 'a' + 1) \times 32^{n-2} + \ldots + (s_{n-1} - 'a' + 1)$

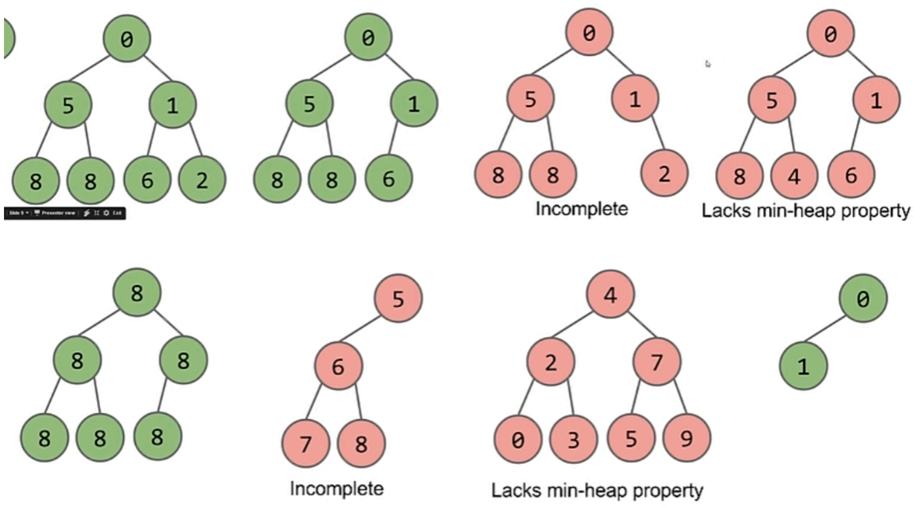
Problems:

- Intended for lower case strings only.
- Top bits are totally ignored.

Lec 24 Priority Queues and Heaps

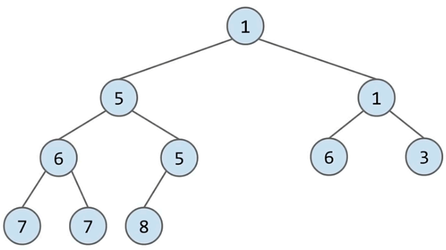
Part 1.Heap

1. **Defination of the Binary min-heap**:Binary tree that is complete and obeys min-heap property
   - ***Min-Heap***:Every node is less than or equal to both of its children.
   - ***Complete***:<u>Missing items only at the bottom level</u>(if any),all nodes are as far as left as possible.

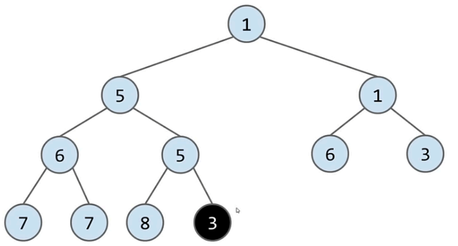


2. Heap operations
   i. **Insertion**

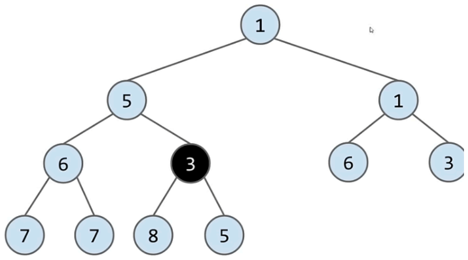
   e.g. Insert 3 into the heap

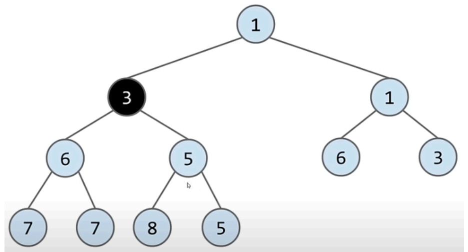


- <u>**step 1**: Add to the end of heap temporarily</u>to maintain the completeness.



- **step 2**: Swim up the hierarchy to your rightful place to keep it a min-heap.

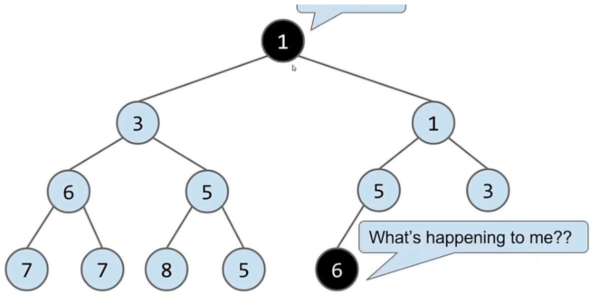


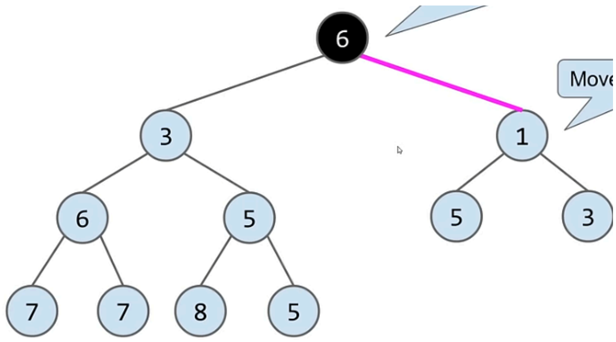


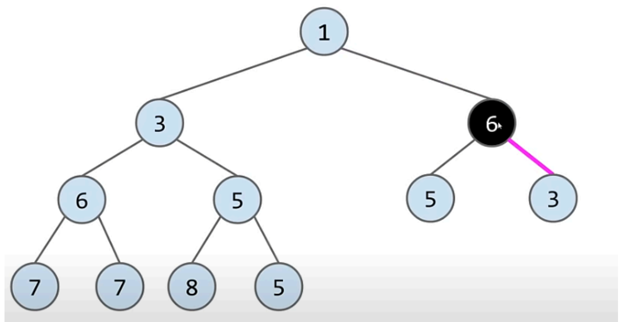
ii. **Delete Min**

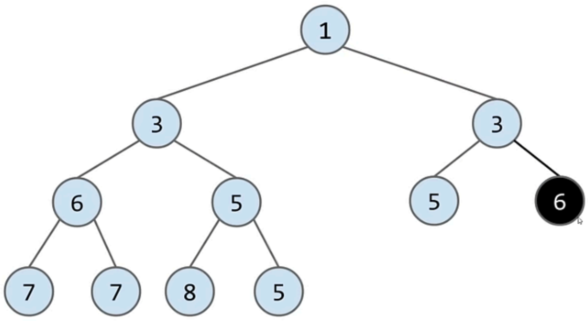
- **step 1**:Swap the last item in the heap into the root
- **step2**:Then sink your way down the hierarchy,yielding to most qualified folks.

Switching the node and its smallest child until it find the rigth position







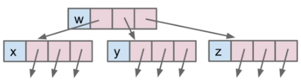


Part 2. Tree Impletation

1. Approch 1:

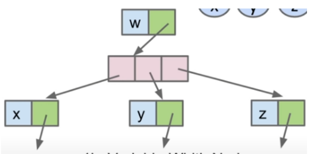
   Create mapping from node to children.

```
/**Approach 1a:Fixed-Width Nodes
 *(BSTMap used this approach)*/
public class Tree1A<Key>{
   Key k;
   Tree1A left;
   Tree1A middle;
   Tree1A right;
}
```



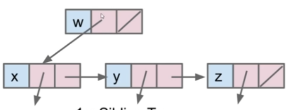
1a: Fixed-Width Nodes (BSTMap used this approach)

```
/*Approach 1b:Variable-Width Nodes*/
public class Tree1B<Key>{
   Key k;
   Tree1B[] children;
}
```



1b: Variable-Width Nodes

```
/*Approach 1c:Sibling(兄弟) Tree*/
//儿子兄弟表示法
public class Tree1C<key>{
   Key k;
   Tree1C favorChild;
   Tree1C sibling;
}
```

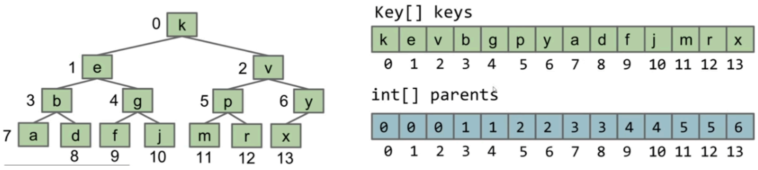


1c: Sibling Tree

2. Approach 2:

   Store keys in an array.Store parentIDs in an array.

```
public class Tree2<Key>{
   Key[] keys;
   int[] parents;
}
```



3. Approach 3:
If we assume that **the tree is complete**,we can even **abandon the parents array** and just **use the index of the Key[] array to compute its parent or children**

```
/*Approach 3a:The index 0 is the root*/
public int parent(int k){
   return (k-1)/2;
}
```

Approach 3b:improvement of approach 3a to make the computation of index easier.The index 0 is the sentinel and the index 1 is the root.

> Suppose the index of the node is x:
> Its parent's index:**x/2**
> Its left Child's index:**2x**
> Its right Child's index:**2x+1**

```
/*The index 0 is the sentinel and the index 1 is the root*/
```

Part 3.Priority Queues

Why should we create "priority queue"?
Because <u>It's hard to handle items with same priority at a time when we use BST.</u>

1. Priority Queue Interface

    ```
    /**(Min) Priority Queue:Allowing tracking and
    removal of the smallest item in a priority queue.*/

    public interface MinPQ<Item> extends Comparable<Item>{

     /*Add the item to the priority queue.*/
     public void ass(Item x);

     /*Returns the smallest item from the priority queue.*/
     public Item getSmallest();

     /*Removes the smallest item from the priority queue.*/
     public Item removeSmallest();

     /*Returns the size of the priority queue.*/
     publi c int size();
    }
    ```

2. We can use the heap to implement the priority queue.

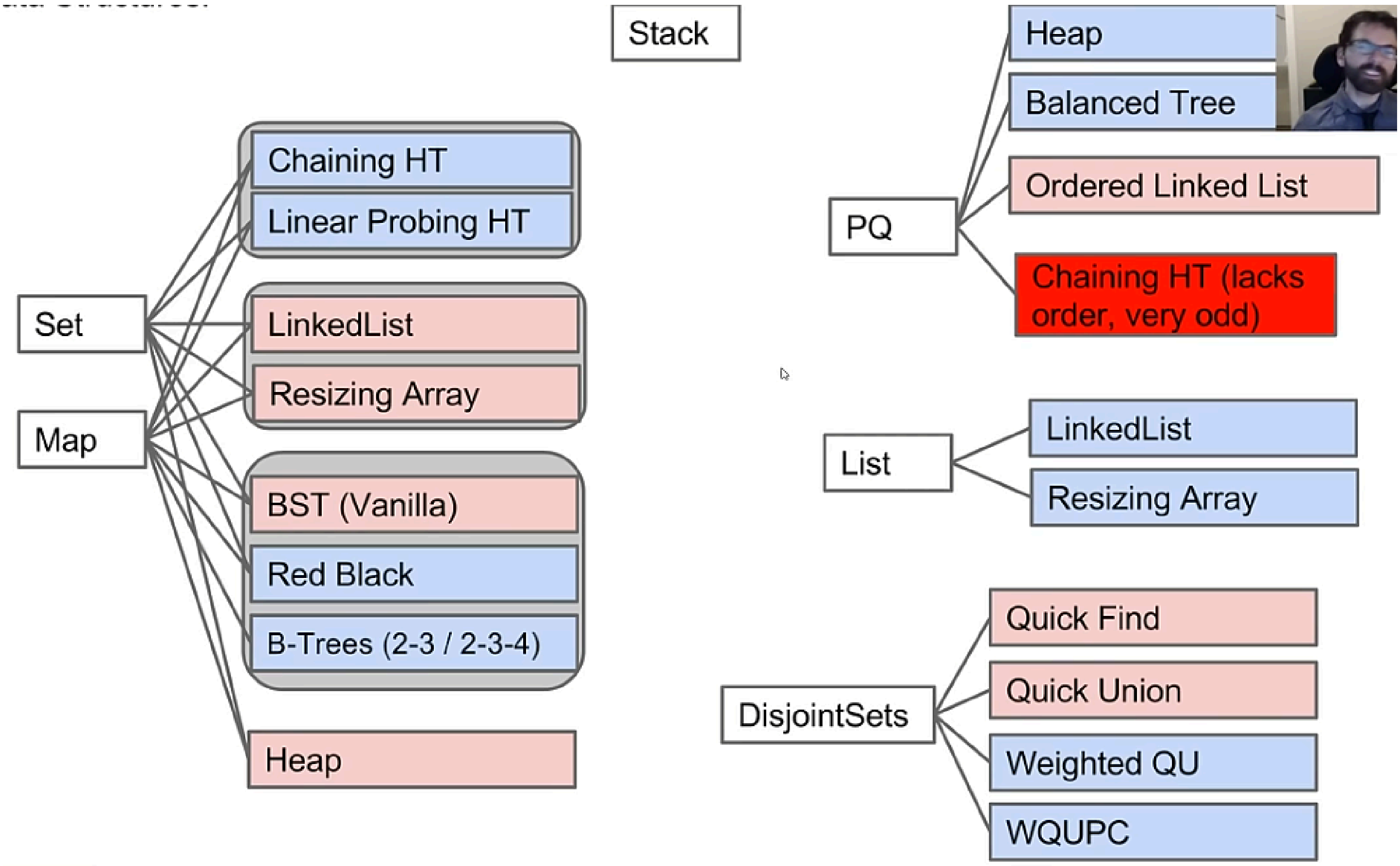
Part 4.Data Structure Comparison

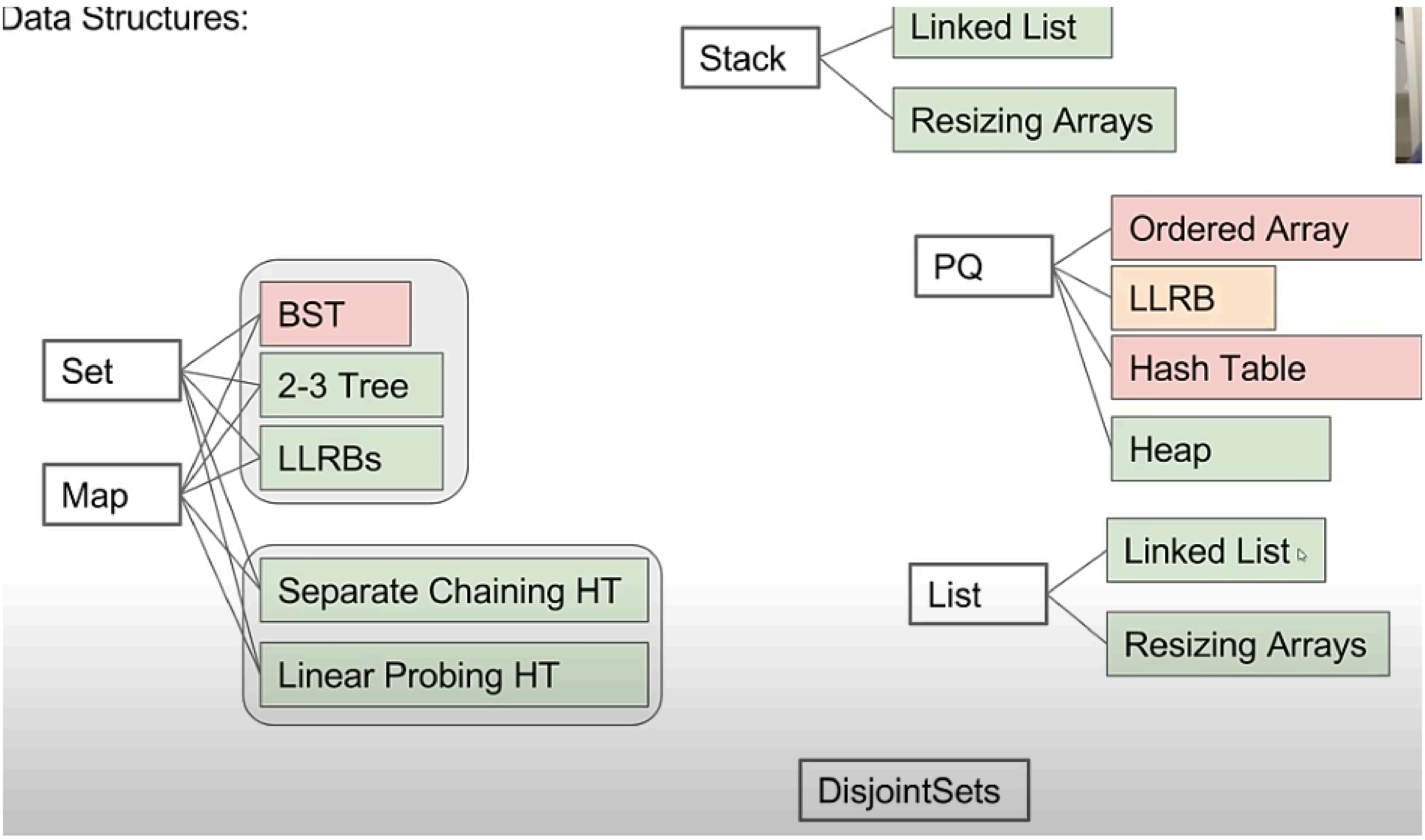
The comparasion of time complexity between different data structures:

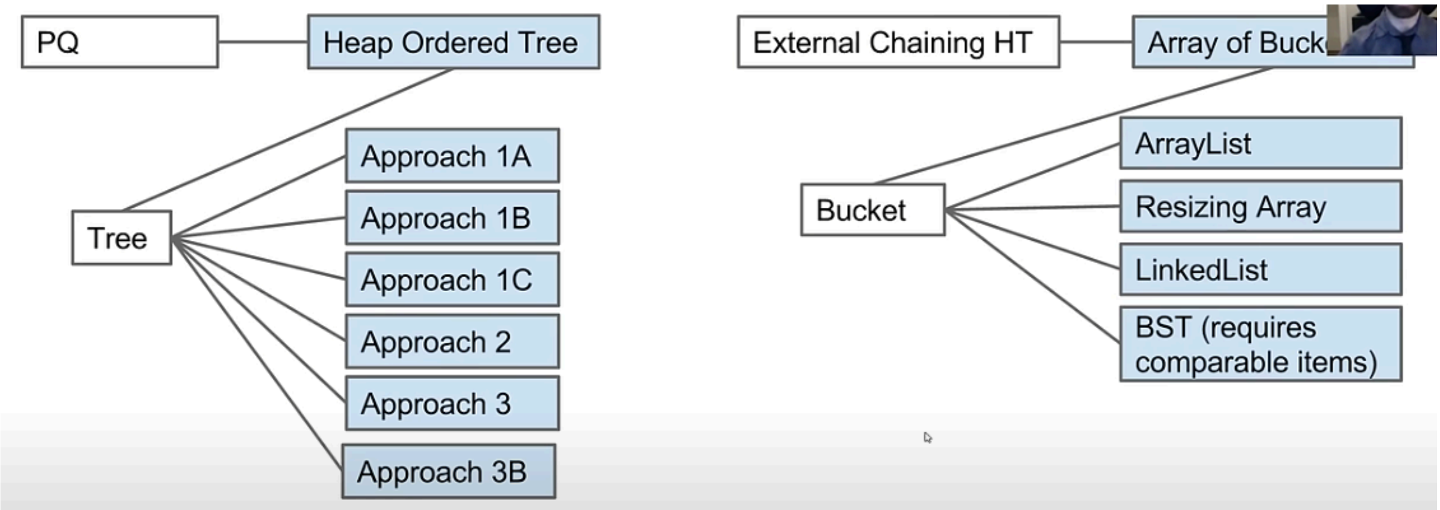
	Ordered Array	Bushy BST	Hash Table	Heap
add	Θ(N)	Θ(log N)	Θ(1)	Θ(log N)
getSmallest	Θ(1)	Θ(log N)	Θ(N)	Θ(1)
removeSmallest	Θ(N)	Θ(log N)	Θ(N)	Θ(log N)

other comparision

//wait to be finished







Lec 25 Tree Traversals ,Quad Trees

Part 1.Tree Traversals

Given a tree,we may need to browse all of its node in some way.
There are two general ways of tree traversals.

- **Depth First Traversal**
  深度优先遍历
  - *PreOrder Traversal*
  - *InOrder Traversal*
  - *PostOrder Trversal*
- **Level Order Traversal(Width First Traversal)**
  广度优先遍历（层序遍历）

Part 2.Depth First Traversal

1. PreOrder Traversal

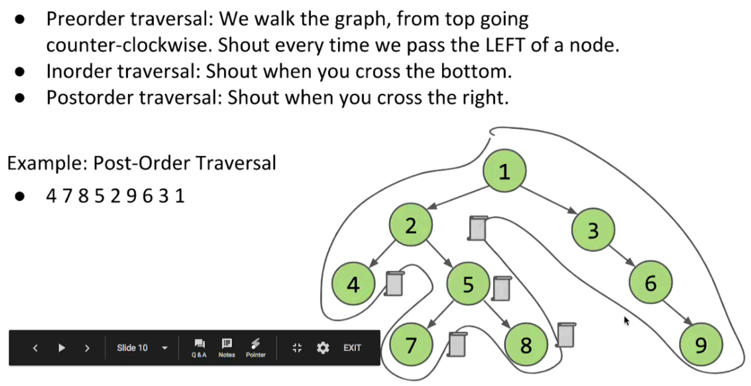
```
preOrder(TreeNode x){
   if(x == Null) return;
   print(x.key);
   preorder(x.left);
   preOrder(x.right);
}
```

2. InOrder Traversal

```
inOrder(TreeNode x){
  if(x == Null) return;
  inorder(x.left);
  print(x.key);
  preOrder(x.right);
}
```

3. PostOrder Traversal

```
postOrder(TreeNode x){
  if(x == Null) return;
  postorder(x.left);
  postOrder(x.right);
  print(x.key);
}
```



Time Complexity of Depth First Traversal:$\Theta(N)$.To traverse every node for 3 times.

Part 3.Level Order Traversal

```
public void levelOrder(Tree T,Action toDo){
   for (int i = 0;i < T.height(); i+=1){
      visitLevel(T, i, toDo);
   }
}


public void visitLevel(Tree T,int level,Action toDo){
   if (T == null){
      return;
   }
   if(lev == 0){
      toDo.visit(T.key);
   }
   else{
      visitLevel(T.left(),lev - 1,toDo);
      visitLevel(T.right(),lev - 1,toDo);
   }
}
```

Time Complexity of level order traversal of a complete Tree:$\Theta(N)$

- The top level considered:1
- The top level 2 levels considered:$1 + 2 = 2^2 - 1$
- The top level 3 levels considered:$1 + 2 + 4 = 2^3 - 1$

  ...
- The top k levels considered:$1 + 2 + 4 + ... + 2^{k-1} = 2^k - 1$

  ...

For the Tree whose height is $H = log_2(N+1)$,the total times of consideration=$2^1 + 2^2 + 2^3 + ... + 2^H - H = 2^{H+1} - 2 - H = 2N - log_2(N+1) = \Theta(N)$

Part 4.Range Finding

Suppose we want an operation that returns all items in an range.

- The **pruning** idea:

  Restricting our search to only nodes that might contain the answers we seek.
- Runtime for our search:$\Theta(logN + R)$

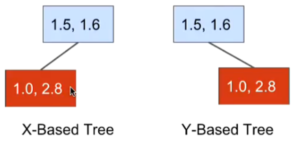
  N:Total number of items in tree

  R:Number of matches

Part 5.Spatial Trees

Some data is two dimentional,e.g. the location of Planets.

- earth.xPos=1.5,earth.yPos=1.6
- Mars.xPos=1.0,Mars.yPos=2.8
  For the xPos,Mars<Earth
  For the yPos,Mars>Earth



1. Quadtree:
   we devide and conquer by splitting 2D space into four quadrants
   - we store items into appropriate quadrant
   - Repeat recursively if more than one item in a quadrant
     defination:
     The quadtree is either:
     - Empty
     - a 'Root' item at position(x,y) and four quadtrees in four quadrant(象限)
     - use two compares to decide which direction to go