The 3 main operations that we perform on data structures:

- **SEARCH**
- **INSERT**
- **DELETE**

How fast can we do these?

### Array

- **SEARCH**
  - by key: $O(n)$
  - if sorted: $O(\log n)$
  - by index: $O(1)$

### Linked List

- **SEARCH**: $O(n)$

### Hash Table

- **SEARCH**:
  - expected: $O(1)$
  - worst-case: $O(n)$

- **INSERT**: $O(1) \rightarrow O(n)$

- **DELETE**: $O(1) \rightarrow O(n)$
Is Hash Table always used?

Example: Let's store memory address intervals in a hash table and find corresponding users.

<table>
<thead>
<tr>
<th>intervals</th>
<th>userids</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,4)</td>
<td>1111</td>
</tr>
<tr>
<td>(6,7)</td>
<td>0e54</td>
</tr>
<tr>
<td>(10,16)</td>
<td>1r22</td>
</tr>
<tr>
<td>(20,22)</td>
<td>jrg94</td>
</tr>
<tr>
<td>(30,36)</td>
<td>r133</td>
</tr>
<tr>
<td>(42,45)</td>
<td>adw58</td>
</tr>
<tr>
<td>(48,50)</td>
<td>rr44</td>
</tr>
</tbody>
</table>

Keys: intervals

Values: userids

Who owns 49?

Hash it up!
The 3 main operations that we perform on data structures are:

- **SEARCH**
  - Array: $O(n)$ (unsorted), $O(\log n)$ (if sorted), $O(1)$ by index
  - Linked List: $O(n)$
  - Hash Table: $O(1) \rightarrow O(n)$

- **INSERT**
  - Array: $O(n)$
  - Linked List: $O(1) \rightarrow O(n)$ (if sorted)
  - Hash Table: $O(1) \rightarrow O(n)$

- **DELETE**
  - Array: $O(n)$
  - Linked List: $O(n)$
  - Hash Table: $O(1) \rightarrow O(n)$
Linked list in a Binary Tree form which does Binary Search.
Trees: Filesystem
BST data structure

- extension of BinaryTree
- invariant:
  - nodes in LEFT subtree are less than root
  - nodes in RIGHT subtree are greater than or equal to root
Given array of elements: 3 1 8 2 6 7 5
Given array of elements: 3 1 8 2 6 7 5
Given array of elements: 3 1 8 2 6 7 5
Given array of elements: 3 1 8 2 6 7 5
Given array of elements: 3 1 8 2 6 7 5
Given array of elements: 3 1 8 2 6 7 5
Given array of elements: 3 1 8 2 6 7 5
**Binary Search Trees**

(binary) search for 5

```
struct node {
  int key;

  struct node *left; /* left child */
  struct node *right; /* right child */
};

#define NUM_CHILDREN (2)
struct node {
  int key;
  struct node *child[NUM_CHILDREN];
};
```
Binary search trees

(binary) search for 5
/* returns pointer to node with given target key */
/* or 0 if no such node exists */

struct node *
treeSearch(struct node *root, int target)
{
    if (root == 0 || root->key == target) {
        return root;
    } else if (root->key > target) {
        return treeSearch(root->left, target);
    } else {
        return treeSearch(root->right, target);
    }
}
Binary search trees

(binary) search for 5

found

time: \(O(\text{depth}) = O(\log n)\)
Binary search trees

(binary) search for 5

Time: \( O(\text{depth}) \backslash O(\log n) \)

(binary) search for 4

\( \times \) not found
Binary Search Trees

Insert (4) - "in" search

Diagram of a binary search tree with nodes labeled 1, 2, 3, 4, 5, 6, 7, and 8.
True or False: nodes that are inserted will always become leaves in the tree
True or False: nodes that are inserted will always become leaves in the tree
Binary search trees

Insert (4) and search

Instant delete (x)

x: leaf
Binary search trees

insert (4) \lor search

instant delete (x)

x: leaf

x: root w/ 1 child

\Rightarrow x = \text{min or max}
Binary Search Trees

Insert (4) vs search

Instant delete (x)

x: leaf

x: root w/ 1 child
\[ \Rightarrow x = \text{MIN or MAX} \]

x: any node w/ 1 child
Binary search trees

**Insert (4)** ~ **Search**

\[
\begin{array}{c}
\text{1} \\
\text{2} \\
\text{3} \\
\text{4} \\
\text{5} \\
\text{6} \\
\text{7} \\
\text{8}
\end{array}
\]

\[O(\text{depth})\]

\[O(\log n)\]

**Instant delete (\(x\))**

\(x\): any node with \(\leq 2\) children

If (one) subtree exists, promote it.
Binary search trees

delete(3)
**Binary Search Trees**

Non-instant delete (3) \[\rightarrow\] find successor: smallest element greater than 3 (which exists because: 2 children)
**Binary Search Trees**

delete(3) → find successor & replace
Binary Search Trees

`delete(3)` → find successor & replace

By definition, successor is the last node visited on a path from R-child(3)
**Binary Search Trees**

delete(3) → find successor & replace

By definition, successor is the last node visited on a path from R-child(3)
struct node* deleteNode(struct node* root, int key) {
    // base case
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key?
    else {
        // node with only one child or no child:
        if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;
        }
        // node with two children:
        struct node* temp = minValueNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;

    // node with only one child or no child:
    if (root->left == NULL) {
        struct node* temp = root->right;
        free(root);
        return temp;
    }
    else if (root->right == NULL) {
        struct node* temp = root->left;
        free(root);
        return temp;
    }
    // node with two children:
    struct node* temp = minValueNode(root->right);
    root->key = temp->key;
    root->right = deleteNode(root->right, temp->key);
}
return root;
```c
struct node* minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current && current->left != NULL)
        current = current->left;

    return current;
}
```
Tree Traversals

Three ways:

**Pre-order**: visit node, left subtree, right subtree.

**In-order**: visit left subtree, node, right subtree.

**Post-order**: visit left subtree, right subtree, node.

(think of the “visit” as a print operation)
void traverse(struct node* root)
{
    if (root != NULL) {
        traverse(root->left);
        printf("%s", root->key);
        traverse(root->right);
    }
}

In-order: Bashful, Doc, Grumpy, Happy, Sleepy, Sneezy
Pre-order: Happy, Doc, Bashful, Grumpy, Sleepy, Sneezy
Post-order: Bashful, Grumpy, Doc, Sneezy, Sleepy, Happy
Binary search trees - built randomly

Insert n elements into a BST in the order that they're given.
Binary Search Tree Summary

Insert, Search, Delete

Search $\rightarrow O(\text{depth})$  $O(\log n)$

We should keep the tree balanced as much as possible.
• What is the worst-case time complexity, and why?
• How unbalanced could the tree be?
• What is the worst-case time complexity, and why?
• How unbalanced could the tree be?

→ already sorted input, reverse-sorted, nearly sorted...

1 2 3 4 5 → 1 1 2 2 3 3 4 4 5
• What is the worst-case time complexity, and why?
• How unbalanced could the tree be?

\[
\begin{array}{c}
1 & 2 & 3 & 4 & 5 \\
\rightarrow & \text{already sorted input, reverse-sorted, nearly sorted...} \\
\end{array}
\]

\[
\begin{array}{c}
1 \quad 1 \quad 1 \quad 1 \\
2 \quad 2 \quad 2 \quad 2 \\
3 \quad 3 \quad 3 \quad 3 \\
4 \quad 4 \quad 4 \quad 4 \\
5 \\
\end{array}
\]

\[O(n) \text{ depth}\]
• What is the worst-case time complexity, and why?
• How unbalanced could the tree be?

\[ \rightarrow \] already sorted input, reverse-sorted, nearly sorted...

1 2 3 4 5 \rightarrow 1 1 1 2 2 2 3 3 3 4 4 4 5

\( O(n) \) depth

• What would be ideal?

\( O(\log n) \)