FR-0: Big-Oh Notation

\( O(f(n)) \) is the set of all functions that are bound from above by \( f(n) \)

\( T(n) \in O(f(n)) \) if

\[ \exists c, n_0 \text{ such that } T(n) \leq c \times f(n) \text{ when } n > n_0 \]

FR-1: Big-Oh Examples

\[ n \in O(n) ? \]
\[ 10n \in O(n) ? \]
\[ n \in O(10n) ? \]
\[ n \in O(n^2) ? \]
\[ n^2 \in O(n) ? \]
\[ 10n^2 \in O(n^2) ? \]
\[ n \log n \in O(n^2) ? \]
\[ \log n \in O(2n) ? \]
\[ \log n \in O(n) ? \]
\[ 3n + 4 \in O(n) ? \]
\[ 5n^2 + 10n - 2 \in O(n^3) ? O(n^2) ? O(n) ? \]

FR-2: Big-Oh Examples

\[ n \in O(n) \]
\[ 10n \in O(n) \]
\[ n \in O(10n) \]
\[ n \in O(n^2) \]
\[ n^2 \notin O(n) \]
\[ 10n^2 \in O(n^2) \]
\[ n \log n \in O(n^2) \]
\[ \log n \in O(2n) \]
\[ \log n \in O(n) \]
\[ 3n + 4 \in O(n) \]
\[ 5n^2 + 10n - 2 \in O(n^3), \in O(n^2), \notin O(n) ? \]

FR-3: Big-Oh Examples II

\[ \sqrt{n} \in O(n) ? \]
\[ \log n \in O(2^n) ? \]
\[ \log n \in O(n) ? \]
\[ n \log n \in O(n) ? \]
\[ n \log n \in O(n^2) ? \]
\[ \sqrt{n} \in O(\log n) ? \]
\[ \log n \in O(\sqrt{n}) ? \]
\[ n \log n \in O(n^{3/2}) ? \]
\[ n^3 + n \log n + n \sqrt{n} \in O(n \log n) ? \]
\[ n^3 + n \log n + n \sqrt{n} \in O(n^3) ? \]
\[ n^3 + n \log n + n \sqrt{n} \in O(n^4) ? \]

FR-4: Big-Oh Examples II
\[ \sqrt{n} \in O(n) \]
\[ \lg n \in O(2^n) \]
\[ \lg n \in O(n) \]
\[ n \lg n \notin O(n) \]
\[ n \lg n \in O(n^2) \]
\[ \sqrt{n} \notin O(\lg n) \]
\[ \lg n \in O(\sqrt{n}) \]
\[ n \lg n \notin O(n^2) \]
\[ n^3 + n \lg n + n \sqrt{n} \notin O(n \lg n) \]
\[ n^3 + n \lg n + n \sqrt{n} \in O(n^3) \]
\[ n^3 + n \lg n + n \sqrt{n} \in O(n^4) \]

**FR-5: Big-Oh Examples III**

\[ f(n) = \begin{cases} 
  n & \text{for } n \text{ odd} \\
  n^3 & \text{for } n \text{ even} 
\end{cases} \]
\[ g(n) = n^2 \]

\[ f(n) \in O(g(n)) ? \]
\[ g(n) \in O(f(n)) ? \]
\[ n \in O(f(n)) ? \]
\[ f(n) \in O(n^3) ? \]

**FR-6: Big-Oh Examples III**

\[ f(n) = \begin{cases} 
  n & \text{for } n \text{ odd} \\
  n^3 & \text{for } n \text{ even} 
\end{cases} \]
\[ g(n) = n^2 \]

\[ f(n) \notin O(g(n)) \]
\[ g(n) \notin O(f(n)) \]
\[ n \in O(f(n)) \]
\[ f(n) \notin O(n^3) \]

**FR-7: Big-Ω Notation**

\[ \Omega(f(n)) \] is the set of all functions that are bound from below by \( f(n) \)

\[ T(n) \in \Omega(f(n)) \text{ if } \]

\[ \exists c, n_0 \text{ such that } T(n) \geq c \cdot f(n) \text{ when } n > n_0 \]

**FR-8: Big-Ω Notation**

\[ \Omega(f(n)) \] is the set of all functions that are bound from below by \( f(n) \)

\[ T(n) \in \Omega(f(n)) \text{ if } \]

\[ \exists c, n_0 \text{ such that } T(n) \geq c \cdot f(n) \text{ when } n > n_0 \]

\[ f(n) \in O(g(n)) \Rightarrow g(n) \in \Omega(f(n)) \]

**FR-9: Big-Θ Notation**

\[ \Theta(f(n)) \] is the set of all functions that are bound both above and below by \( f(n) \). \( \Theta \) is a tight bound

\[ T(n) \in \Theta(f(n)) \text{ if } \]
\[ T(n) \in O(f(n)) \text{ and } T(n) \in \Omega(f(n)) \]

FR-10: **Big-Oh Rules**

1. If \( f(n) \in O(g(n)) \) and \( g(n) \in O(h(n)) \), then \( f(n) \in O(h(n)) \)
2. If \( f(n) \in O(kg(n)) \) for any constant \( k > 0 \), then \( f(n) \in O(g(n)) \)
3. If \( f_1(n) \in O(g_1(n)) \) and \( f_2(n) \in O(g_2(n)) \), then \( f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n))) \)
4. If \( f_1(n) \in O(g_1(n)) \) and \( f_2(n) \in O(g_2(n)) \), then \( f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n)) \)

(Also work for \( \Omega \), and hence \( \Theta \))

FR-11: **Big-Oh Guidelines**

- Don’t include constants/low order terms in Big-Oh
- Simple statements: \( \Theta(1) \)
- Loops: \( \Theta(\text{inside}) \cdot \# \text{ of iterations} \)
  - Nested loops work the same way
- Consecutive statements: Longest Statement
- Conditional (if) statements:
  \( O(\text{Test} + \text{longest branch}) \)

FR-12: **Calculating Big-Oh**

```c
for (i=1; i<n; i++)
  for (j=1; j < n/2; j++)
    sum++;
```

FR-13: **Calculating Big-Oh**

```c
for (i=1; i<n; i++)
  for (j=1; j < n/2; j++)
    sum++; Executed n times
  sum++; Executed n/2 times
```

Running Time: \( O(n^2), \Omega(n^2), \Theta(n^2) \)

FR-14: **Calculating Big-Oh**

```c
for (i=1; i<n; i=i*2)
  sum++;
```

FR-15: **Calculating Big-Oh**

```c
for (i=1; i<n; i=i*2)
  sum++; Executed \lg n \times 1 \text{ times}
```

Running Time: \( O(\lg n), \Omega(\lg n), \Theta(\lg n) \)

FR-16: **Calculating Big-Oh**

```c
for (i=1; i<n; i=i*2)
  sum++;
```
for (i=1; i<n; i=i*2)
    for (j=0; j < n; j = j + 1)
        sum++;
for (i=n; i >1; i = i / 2)
    for (j = 1; j < n; j = j * 2)
        for (k = 1; k < n; k = k * 3)
            sum++

FR-17: Recurrence Relations

$T(n) =$ Time required to solve a problem of size $n$

Recurrence relations are used to determine the running time of recursive programs – recurrence relations themselves are recursive

$T(0) =$ time to solve problem of size 0
  – Base Case
$T(n) =$ time to solve problem of size $n$
  – Recursive Case

FR-18: Recurrence Relations

long power(long x, long n) {
    if (n == 0) return 1;
    else return x * power(x, n-1);
}

$T(0) =$ $c_1$ for some constant $c_1$
$T(n) =$ $c_2 + T(n-1)$ for some constant $c_2$

FR-19: Building a Better Power

long power(long x, long n) {
    if (n==0) return 1;
    if (n==1) return x;
    if ((n % 2) == 0)
        return power(x*x, n/2);
    else
        return power(x*x, n/2) * x;
}

FR-20: Building a Better Power

long power(long x, long n) {
    if (n==0) return 1;
    if (n==1) return x;
    if ((n % 2) == 0)
        return power(x*x, n/2);
    else
        return power(x*x, n/2) * x;
}

$T(0) =$ $c_1$
$T(1) =$ $c_2$
$T(n) =$ $T(n/2) + c_3$
FR-21: Solving Recurrence Relations

\[ T(n) = T(n/2) + c_3 \]
\[ = T(n/4) + c_3 + c_3 \]
\[ = T(n/4)2c_3 \]
\[ = T(n/8) + c_3 + 2c_3 \]
\[ = T(n/8)3c_3 \]
\[ = T(n/16) + c_3 + 3c_3 \]
\[ = T(n/16) + 4c_3 \]
\[ = T(n/32) + c_3 + 4c_3 \]
\[ = T(n/32) + 5c_3 \]
\[ \cdots \]
\[ = T(n/2^k) + kc_3 \]

\[ T(0) = c_1 \]
\[ T(1) = c_2 \]
\[ T(n) = T(n/2) + c_3 \]
\[ T(n) = T(n/2^k) + kc_3 \]

We want to get rid of \( T(n/2^k) \). Since we know \( T(1) \) ...

\[
\begin{align*}
  n/2^k & = 1 \\
n & = 2^k \\
lg n & = k
\end{align*}
\]

FR-22: Solving Recurrence Relations

\[ T(1) = c_2 \]
\[ T(n) = T(n/2^k) + kc_3 \]

\[
\begin{align*}
  T(n) & = T(n/2^{\lg n}) + \lg nc_3 \\
        & = T(1) + c_3 \lg n \\
        & = c_2 + c_3 \lg n \\
        & \in \Theta(\lg n)
\end{align*}
\]

FR-23: Abstract Data Types

- An Abstract Data Type is a definition of a type based on the operations that can be performed on it.
- An ADT is an interface
- Data in an ADT cannot be manipulated directly – only through operations defined in the interface

FR-24: Stack

A Stack is a Last-In, First-Out (LIFO) data structure.

Stack Operations:
- Add an element to the top of the stack
- Remove the top element
Check if the stack is empty

FR-26: **Stack Implementation**

Array:

- Stack elements are stored in an array
- Top of the stack is the *end* of the array
  - If the top of the stack was the beginning of the array, a push or pop would require moving all elements in the array
- Push: data[top++] = elem
- Pop: elem = data[--top]

FR-27: **Stack Implementation**

Linked List:

- Stack elements are stored in a linked list
- Top of the stack is the *front* of the linked list
- push: top = new Link(elem, top)
- pop: elem = top.element(); top = top.next()

FR-28: **Queue**

A Queue is a Last-In, First-Out (FIFO) data structure.

Queue Operations:

- Add an element to the end (tail) of the Queue
- Remove an element from the front (head) of the Queue
- Check if the Queue is empty

FR-29: **Queue Implementation**

Linked List:

- Maintain a pointer to the first and last element in the Linked List
- Add elements to the back of the Linked List
- Remove elements from the front of the linked list
  - Enqueue: tail.setNext(new link(elem,null));
    tail = tail.next();
  - Dequeue: elem = head.element();
    head = head.next();

FR-30: **Queue Implementation**

Array:

- Store queue elements in a circular array
- Maintain the index of the first element (head) and the next location to be inserted (tail)
- Enqueue: \( \text{data[tail]} = \text{elem}; \)
  \[\text{tail} = (\text{tail} + 1) \mod \text{size}\]
- Dequeue: \( \text{elem} = \text{data[head]}; \)
  \[\text{head} = (\text{head} + 1) \mod \text{size}\]

FR-31: Binary Trees
Binary Trees are Recursive Data Structures
- Base Case: Empty Tree
- Recursive Case: Node, consisting of:
  - Left Child (Tree)
  - Right Child (Tree)
  - Data

FR-32: Binary Tree Examples
The following are all Binary Trees (Though not Binary Search Trees)

FR-33: Tree Terminology
- Parent / Child
- Leaf node
- Root node
- Edge (between nodes)
- Path
- Ancestor / Descendant
- Depth of a node \( n \)
  - Length of path from root to \( n \)
- Height of a tree
• (Depth of deepest node) + 1

FR-34: Binary Search Trees

• Binary Trees
  • For each node n, \((\text{value stored at node n}) > (\text{value stored in left subtree})\)
  • For each node n, \((\text{value stored at node n}) < (\text{value stored in right subtree})\)

FR-35: Writing a Recursive Algorithm

• Determine a small version of the problem, which can be solved immediately. This is the base case
• Determine how to make the problem smaller
• Once the problem has been made smaller, we can assume that the function that we are writing will work correctly on the smaller problem (Recursive Leap of Faith)
  • Determine how to use the solution to the smaller problem to solve the larger problem

FR-36: Finding an Element in a BST

• First, the Base Case – when is it easy to determine if an element is stored in a Binary Search Tree?
  • If the tree is empty, then the element can’t be there
  • If the element is stored at the root, then the element is there

FR-37: Finding an Element in a BST

• Next, the Recursive Case – how do we make the problem smaller?
  • Both the left and right subtrees are smaller versions of the problem. Which one do we use?
  • If the element we are trying to find is \(<\) the element stored at the root, use the left subtree. Otherwise, use the right subtree.
  • How do we use the solution to the subproblem to solve the original problem?
    • The solution to the subproblem is the solution to the original problem (this is not always the case in recursive algorithms)

FR-38: Printing out a BST

To print out all element in a BST:

• Print all elements in the left subtree, in order
• Print out the element at the root of the tree
• Print all elements in the right subtree, in order
  • Each subproblem is a smaller version of the original problem – we can assume that a recursive call will work!

FR-39: Printing out a BST
```java
void print(Node tree) {
    if (tree != null) {
        print(tree.left());
        System.out.println(tree.element());
        print(tree.right());
    }
}

FR-40: Inserting e into BST T

- Base case – T is empty:
  - Create a new tree, containing the element e
- Recursive Case:
  - If e is less than the element at the root of T, insert e into left subtree
  - If e is greater than the element at the root of T, insert e into the right subtree

Node insert(Node tree, Comparable elem) {
    if (tree == null) {
        return new Node(elem);
    }
    if (elem.compareTo(tree.element()) < 0) {
        tree.setLeft(insert(tree.left(), elem));
        return tree;
    } else {
        tree.setRight(insert(tree.right(), elem));
        return tree;
    }
}

FR-41: Inserting e into BST T

FR-42: Deleting From a BST

- Removing a leaf:
  - Remove element immediately
- Removing a node with one child:
  - Just like removing from a linked list
  - Make parent point to child
- Removing a node with two children:
  - Replace node with largest element in left subtree, or the smallest element in the right subtree

FR-43: Priority Queue ADT

Operations

- Add an element / priority pair
- Return (and remove) element with highest priority
Implementation:

- Heap
  - Add Element \( O(lg n) \)
  - Remove Highest Priority \( O(lg n) \)

**FR-44: Heap Definition**

- Complete Binary Tree
- Heap Property
  - For every subtree in a tree, each value in the subtree is \( \geq \) value stored at the root of the subtree

**FR-45: Heap Examples**

```
1
  2
 / \
7  3
 /   \
9  8  5  4
```

Valid Heap

**FR-46: Heap Examples**

```
1
  8
 / \
2 9
 /   \
5 7 10 13
```

Invalid Heap

**FR-47: Heap Insert**

- There is only one place we can insert an element into a heap, so that the heap remains a complete binary tree
- Inserting an element at the “end” of the heap might break the heap property
  - Swap the inserted value up the tree
FR-48: **Heap Remove Largest**

- Removing the Root of the heap is hard
- Removing the element at the “end” of the heap is easy
  - Move last element into root
  - Shift the root down, until heap property is satisfied

FR-49: **Representing Heaps**

A Complete Binary Tree can be stored in an array:

```
1
  2       14
  5  3    16  15
 7  6  8  9
0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

FR-50: **CBTs as Arrays**

- The root is stored at index 0
- For the node stored at index $i$:
  - Left child is stored at index $2 \times i + 1$
  - Right child is stored at index $2 \times i + 2$
  - Parent is stored at index $\lfloor (i - 1)/2 \rfloor$

FR-51: **Trees with > 2 children**

How can we implement trees with nodes that have > 2 children?
FR-52: **Trees with > 2 children**

- Array of Children

![Diagram of Array of Children]

FR-53: **Trees with > 2 children**

- Linked List of Children

![Diagram of Linked List of Children]

FR-54: **Left Child / Right Sibling**

- We can integrate the linked lists with the nodes themselves:
FR-55: **Serializing Binary Trees**

- Printing out nodes, in order that they would appear in a PREORDER traversal does not work, because we don’t know when we’ve hit a null pointer
- Store null pointers, too!

FR-56: **Serializing Binary Trees**

- In most trees, more null pointers than internal nodes
- Instead of marking null pointers, mark internal nodes
- Still need to mark some nulls, for nodes with 1 child

FR-57: **Serializing General Trees**
• Store an “end of children” marker

![Tree Diagram]

FR-58: **Main Memory Sorting**

• All data elements can be stored in memory at the same time
• Data stored in an array, indexed from 0 \( \ldots n - 1 \), where \( n \) is the number of elements
• Each element has a key value (accessed with a `key()` method)
• We can compare keys for \( \lt, =, \gt \)
• For illustration, we will use arrays of integers – though often keys will be strings, other Comparable types

FR-59: **Stable Sorting**

• A sorting algorithm is *Stable* if the relative order of duplicates is preserved
• The order of duplicates matters if the keys are duplicated, but the *records* are not.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>B</td>
<td>J</td>
<td>E</td>
<td>A</td>
<td>S</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Data</td>
<td>o</td>
<td>o</td>
<td>d</td>
<td>m</td>
<td>u</td>
<td>l</td>
<td>u</td>
</tr>
<tr>
<td>Data</td>
<td>b</td>
<td>e</td>
<td></td>
<td>y</td>
<td>e</td>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>

A *non*-Stable sort

FR-60: **Insertion Sort**

• Separate list into sorted portion, and unsorted portion
• Initially, sorted portion contains first element in the list, unsorted portion is the rest of the list
  • (A list of one element is always sorted)
• Repeatedly insert an element from the unsorted list into the sorted list, until the list is sorted
FR-61: **Bubble Sort**

- Scan list from the last index to index 0, swapping the smallest element to the front of the list
- Scan the list from the last index to index 1, swapping the second smallest element to index 1
- Scan the list from the last index to index 2, swapping the third smallest element to index 2
  ... 
- Swap the second largest element into position \(n - 2\)

FR-62: **Selection Sort**

- Scan through the list, and find the smallest element
- Swap smallest element into position 0
- Scan through the list, and find the second smallest element
- Swap second smallest element into position 1
  ... 
- Scan through the list, and find the second largest element
- Swap smallest largest into position \(n - 2\)

FR-63: **Shell Sort**

- Sort \(n/2\) sublists of length 2, using insertion sort
- Sort \(n/4\) sublists of length 4, using insertion sort
- Sort \(n/8\) sublists of length 8, using insertion sort
  ... 
- Sort 2 sublists of length \(n/2\), using insertion sort
- Sort 1 sublist of length \(n\), using insertion sort

FR-64: **Merge Sort**

- **Base Case:**
  - A list of length 1 or length 0 is already sorted
- **Recursive Case:**
  - Split the list in half
  - Recursively sort two halves
  - Merge sorted halves together

Example: 5 1 8 2 6 4 3 7  FR-65: **Divide & Conquer**

Quick Sort:

- Divide the list two parts
  - Some work required – Small elements in left sublist, large elements in right sublist
• Recursively sort two parts
• Combine sorted lists into one list
  • No work required!

FR-66: **Quick Sort**

• Pick a pivot element
• Reorder the list:
  • All elements < pivot
  • Pivot element
  • All elements > pivot
• Recursively sort elements < pivot
• Recursively sort elements > pivot

Example: 3 7 2 8 1 4 6

FR-67: **Comparison Sorting**

• Comparison sorts work by comparing elements
  • Can only compare 2 elements at a time
  • Check for <, >, =.
• All the sorts we have seen so far (Insertion, Quick, Merge, Heap, etc.) are comparison sorts
• If we know nothing about the list to be sorted, we need to use a comparison sort

FR-68: **Sorting Lower Bound**

• All comparison sorting algorithms can be represented by a decision tree with \( n! \) leaves
• Worst-case number of comparisons required by a sorting algorithm represented by a decision tree is the height of the tree
• A decision tree with \( n! \) leaves must have a height of at least \( n \log n \)
• All comparison sorting algorithms have worst-case running time \( \Omega(n \log n) \)

FR-69: **Binsort**

• Sort \( n \) elements, in the range 1 . . . \( m \)
• Keep a list of \( m \) linked lists
• Insert each element into the appropriate linked lists
• Collect the lists together

FR-70: **Bucket Sort**

• Modify binsort so that each list can hold a range of values
• Need to keep each bucket sorted
FR-71: **Counting Sort**

```java
for (i=0; i<A.length; i++)
    C[A[i].key()]++;

for (i=1; i<C.length; i++)
    C[i] = C[i] + C[i-1];

for (i=A.length - 1; i>=0; i--)
    C[A[i].key()]--;

for (i=0; i<A.length; i++)
    A[i] = B[i];
```

FR-72: **Radix Sort**

- Sort a list of numbers one digit at a time
  - Sort by 1st digit, then 2nd digit, etc
  - Each sort can be done in linear time, using counting sort

- First Try: Sort by most significant digit, then the next most significant digit, and so on
  - Need to keep track of a lot of sublists

FR-73: **Radix Sort** Second Try:

- Sort by *least significant* digit first
- Then sort by next-least significant digit, using a Stable sort
  
  ...  

- Sort by most significant digit, using a Stable sort

At the end, the list will be completely sorted.

FR-74: **Searching & Selecting**

- Maintain a Database (keys and associated data)
- Operations:
  - **Add** a key / value pair to the database
  - **Remove** a key (and associated value) from the database
  - **Find** the value associated with a key

FR-75: **Hash Function**

- What if we had a “magic function” –
- Takes a key as input
- Returns the index in the array where the key can be found, if the key is in the array

- To add an element
  - Put the key through the magic function, to get a location
  - Store element in that location

- To find an element
  - Put the key through the magic function, to get a location
  - See if the key is stored in that location

FR-76: Hash Function

- The “magic function” is called a Hash function
- If \( \text{hash}(\text{key}) = i \), we say that the key hashes to the value \( i \)
- We’d like to ensure that different keys will always hash to different values.
  - Not possible – too many possible keys

FR-77: Integer Hash Function

- When two keys hash to the same value, a collision occurs.
- We cannot avoid collisions, but we can minimize them by picking a hash function that distributes keys evenly through the array.
- Example: Keys are integers
  - Keys are in range \( 1 \ldots m \)
  - Array indices are in range \( 1 \ldots n \)
  - \( n < m \)
  - \( \text{hash}(k) = k \mod n \)

FR-78: String Hash Function

- Hash tables are usually used to store string values
- If we can convert a string into an integer, we can use the integer hash function
- How can we convert a string into an integer?
  - Concatenate ASCII digits together

\[
\sum_{k=0}^{\text{keys} - 1} \text{key}[k] 
\times 256^{\text{keys} - k - 1}
\]

FR-79: String Hash Function

- Concatenating digits does not work, since numbers get big too fast. Solutions:
• Overlap digits a little (use base of 32 instead of 256)
• Ignore early characters (shift them off the left side of the string)

```
static long hash(String key, int tablesize) {
    long h = 0;
    int i;
    for (i=0; i<key.length(); i++)
        h = (h << 4) + (int) key.charAt(i);
    return h % tablesize;
}
```

FR-80: **ElfHash**

• For each new character, the hash value is shifted to the left, and the new character is added to the accumulated value.
• If the string is long, the early characters will “fall off” the end of the hash value when it is shifted
  • Early characters will not affect the hash value of large strings
• Instead of falling off the end of the string, the most significant bits can be shifted to the middle of the string, and XOR’ed.
• Every character will influence the value of the hash function.

FR-81: **Collisions**

• When two keys hash to the same value, a collision occurs
• A collision strategy tells us what to do when a collision occurs
• Two basic collision strategies:
  • Open Hashing (Closed Addressing, Separate Chaining)
  • Closed Hashing (Open Addressing)

FR-82: **Closed Hashing**

• To add element X to a closed hash table:
  • Find the smallest i, such that Array[hash(x) + f(i)] is empty (wrap around if necessary)
  • Add X to Array[hash(x) + f(i)]
  • If f(i) = i, linear probing

FR-83: **Closed Hashing**

• Quadratic probing
  • Find the smallest i, such that Array[hash(x) + f(i)] is empty
  • Add X to Array[hash(x) + f(i)]
  • f(i) = i^2

FR-84: **Closed Hashing**
• Multiple keys hash to the same element
  • Secondary clustering

• Double Hashing
  • Use a secondary hash function to determine how far ahead to look
  • \( f(i) = i \times \text{hash2(key)} \)

FR-85: Disjoint Sets

• Elements will be integers (for now)

• Operations:
  • CreateSets(n) – Create n sets, for integers 0..(n-1)
  • Union(x,y) – merge the set containing x and the set containing y
  • Find(x) – return a representation of x’s set
    • Find(x) = Find(y) iff x,y are in the same set

FR-86: Implementing Disjoint Sets

• Find: (pseudo-Java)

```java
int Find(x) {
    while (Parent[x] > 0)
        x = Parent[x]
    return x
}
```

FR-87: Implementing Disjoint Sets

• Union(x,y) (pseudo-Java)

```java
void Union(x,y) {
    rootx = Find(x);
    rooty = Find(y);
    Parent[rootx] = Parent[rooty];
}
```

FR-88: Union by Rank

• When we merge two sets:
  • Have the shorter tree point to the taller tree
  • Height of taller tree does not change
  • If trees have the same height, choose arbitrarily

FR-89: Path Compression

• After each call to Find(x), change x’s parent pointer to point directly at root
• Also, change all parent pointers on path from x to root
FR-90: **Graphs**

- A graph consists of:
  - A set of **nodes** or **vertices** (terms are interchangable)
  - A set of **edges** or **arcs** (terms are interchangable)
- Edges in graph can be either directed or undirected

FR-91: **Graphs & Edges**

- Edges can be labeled or unlabeled
  - Edge labels are typically the *cost* associated with an edge
  - e.g., Nodes are cities, edges are roads between cities, edge label is the length of road

FR-92: **Graph Representations**

- Adjacency Matrix
  - Represent a graph with a two-dimensional array $G$
    - $G[i][j] = 1$ if there is an edge from node $i$ to node $j$
    - $G[i][j] = 0$ if there is no edge from node $i$ to node $j$
  - If graph is undirected, matrix is symmetric
  - Can represent edges labeled with a cost as well:
    - $G[i][j] =$ cost of link between $i$ and $j$
    - If there is no direct link, $G[i][j] = \infty$

FR-93: **Adjacency Matrix**

- Examples:

```
   0 1
  0 0 1
  1 1 0
  2 0 0 0
  3 1 1 0
```

FR-94: **Adjacency Matrix**

- Examples:
FR-95: **Graph Representations**

- **Adjacency List**
  - Maintain a linked-list of the neighbors of every vertex.
    - $n$ vertices
    - Array of $n$ lists, one per vertex
    - Each list $i$ contains a list of all vertices adjacent to $i$.

FR-96: **Adjacency List**

- Examples:

```
0 - 1
  |
  v
2 - 3
```

FR-97: **Adjacency List**

- Examples:

```
0 - 1
  |
  v
2 - 3
```

- Note – lists are not always sorted

FR-98: **Topological Sort**
Directed Acyclic Graph, Vertices $v_1 \ldots v_n$

Create an ordering of the vertices
  
  - If there a path from $v_i$ to $v_j$, then $v_i$ appears before $v_j$ in the ordering
  
  Example: Prerequisite chains

FR-99: **Topological Sort**

- Pick a node $v_k$ with no incident edges
- Add $v_k$ to the ordering
- Remove $v_k$ and all edges from $v_k$ from the graph
- Repeat until all nodes are picked.

FR-100: **Graph Traversals**

- Visit every vertex, in an order defined by the topology of the graph.

  Two major traversals:
  
  - Depth First Search
  - Breadth First Search

FR-101: **Depth First Search**

- Starting from a specific node (pseudo-code):

```java
DFS(Edge G[], int vertex, boolean Visited[]) { 
    Visited[vertex] = true;
    for each node w adjacent to vertex: 
        if (!Visited[w]) 
            DFS(G, w, Visited);
}
```

FR-102: **Depth First Search**

```java
class Edge { 
    public int neighbor;
    public int next;
}
void DFS(Edge G[], int vertex, boolean Visited[]) { 
    Edge tmp;
    Visited[vertex] = true;
    for (tmp = G[vertex]; tmp != null; tmp = tmp.next) { 
        if (!Visited[tmp.neighbor]) 
            DFS(G, tmp.neighbor, Visited);
    }
}
```

FR-103: **Breadth First Search**

- DFS: Look as Deep as possible, before looking wide
• Examine all descendants of a node, before looking at siblings

• BFS: Look as **Wide** as possible, before looking deep
  • Visit all nodes 1 away, then 2 away, then three away, and so on

**FR-104: Search Trees**

• Describes the order that nodes are examined in a traversal

• Directed Tree
  • Directed edge from $v_1$ to $v_2$ if the edge $(v_1, v_2)$ was followed during the traversal

**FR-105: Computing Shortest Path**

• Given a directed weighted graph $G$ (all weights non-negative) and two vertices $x$ and $y$, find the least-cost path from $x$ to $y$ in $G$.
  • Undirected graph is a special case of a directed graph, with symmetric edges
  • Least-cost path may not be the path containing the fewest edges
    • “shortest path” == “least cost path”
    • “path containing fewest edges” = “path containing fewest edges”

**FR-106: Single Source Shortest Path**

• If all edges have unit weight,
  • We can use Breadth First Search to compute the shortest path
  • BFS Spanning Tree contains shortest path to each node in the graph
    • Need to do some more work to create & save BFS spanning tree
  • When edges have differing weights, this obviously will not work

**FR-107: Single Source Shortest Path**

• Divide the vertices into two sets:
  • Vertices whose shortest path from the initial vertex is known
  • Vertices whose shortest path from the initial vertex is not known
  • Initially, only the initial vertex is known
  • Move vertices one at a time from the unknown set to the known set, until all vertices are known

**FR-108: Dijkstra’s Algorithm**

• Keep a table that contains, for each vertex
  • Is the distance to that vertex known?
  • What is the best distance we’ve found so far?

• Repeat:
- Pick the smallest unknown distance
- mark it as known
- update the distance of all unknown neighbors of that node
- Until all vertices are known

FR-109: **Spanning Trees**

- Given a connected, undirected graph $G$
  - A subgraph of $G$ contains a subset of the vertices and edges in $G$
  - A Spanning Tree $T$ of $G$ is:
    - subgraph of $G$
    - contains all vertices in $G$
    - connected
    - acyclic

FR-110: **Spanning Tree Examples**

- Graph

![Graph Image]

FR-111: **Spanning Tree Examples**

- Spanning Tree

![Spanning Tree Image]

FR-112: **Minimal Cost Spanning Tree**
• Minimal Cost Spanning Tree
  • Given a weighted, undirected graph \( G \)
  • Spanning tree of \( G \) which minimizes the sum of all weights on edges of spanning tree

FR-113: **Kruskal’s Algorithm**

• Start with an empty graph (no edges)
• Sort the edges by cost
• For each edge \( e \) (in increasing order of cost)
  • Add \( e \) to \( G \) if it would not cause a cycle

FR-114: **Kruskal’s Algorithm**

• We need to:
  • Put each vertex in its own tree
  • Given any two vertices \( v_1 \) and \( v_2 \), determine if they are in the same tree
  • Given any two vertices \( v_1 \) and \( v_2 \), merge the tree containing \( v_1 \) and the tree containing \( v_2 \)
• ... sound familiar?

FR-115: **Kruskal’s Algorithm**

• Disjoint sets!
• Create a list of all edges
• Sort list of edges
• For each edge \( e = (v_1, v_2) \) in the list
  • if \( \text{FIND}(v_1) \neq \text{FIND}(v_2) \)
    • Add \( e \) to spanning tree
    • \( \text{UNION}(v_1, v_2) \)

FR-116: **Prim’s Algorithm**

• Grow that spanning tree out from an initial vertex
• Divide the graph into two sets of vertices
  • vertices in the spanning tree
  • vertices not in the spanning tree
• Initially, Start vertex is in the spanning tree, all other vertices are not in the tree
  • Pick the initial vertex arbitrarily

FR-117: **Prim’s Algorithm**

• While there are vertices not in the spanning tree
• Add the cheapest vertex to the spanning tree

FR-118: **Indexing**

• Operations:
  • Add an element
  • Remove an element
  • Find an element, using a key
  • Find all elements in a range of key values

FR-119: **2-3 Trees**

• Generalized Binary Search Tree
  • Each node has 1 or 2 keys
  • Each (non-leaf) node has 2-3 children
    • hence the name, 2-3 Trees
  • All leaves are at the same depth

FR-120: **Finding in 2-3 Trees**

• How can we find an element in a 2-3 tree?
  • If the tree is empty, return false
  • If the element is stored at the root, return true
  • Otherwise, recursively find in the appropriate subtree

FR-121: **Inserting into 2-3 Trees**

• Always insert at the leaves
• To insert an element:
  • Find the leaf where the element would live, if it was in the tree
  • Add the element to that leaf
    • What if the leaf already has 2 elements?
      • Split!

FR-122: **Splitting nodes**

• To split a node in a 2-3 tree that has 3 elements:
  • Split nodes into two nodes
    • One node contains the smallest element
    • Other node contains the largest element
  • Add median element to parent
    • Parent can then handle the extra pointer

FR-123: **2-3 Tree Example**
• Inserting elements 1-9 (in order) into a 2-3 tree

FR-124: 2-3 Tree Example
• Inserting elements 1-9 (in order) into a 2-3 tree

FR-125: 2-3 Tree Example
• Inserting elements 1-9 (in order) into a 2-3 tree

FR-126: 2-3 Tree Example
• Inserting elements 1-9 (in order) into a 2-3 tree

FR-127: 2-3 Tree Example
• Inserting elements 1-9 (in order) into a 2-3 tree

FR-128: 2-3 Tree Example
• Inserting elements 1-9 (in order) into a 2-3 tree

Too many keys, need to split
FR-129: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree

```
        2  4
      /   \
     1     5
```

FR-130: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree

```
        2  4
      /   \
     1     5 6
```

FR-131: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree

```
        2  4  6
      /   \
     1     5 6 7
```

```
Too many keys need to split
```

FR-132: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree

```
        2  4  6
      /   \
     1     5 7
```

```
Too many keys need to split
```

FR-133: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree
FR-134: **2-3 Tree Example**
- Inserting elements 1-9 (in order) into a 2-3 tree

FR-135: **2-3 Tree Example**
- Inserting elements 1-9 (in order) into a 2-3 tree

FR-136: **2-3 Tree Example**
- Inserting elements 1-9 (in order) into a 2-3 tree

Too many keys need to split

FR-137: **Deleting Leaves**
• If leaf contains 2 keys
  - Can safely remove a key

FR-138: Deleting Leaves

[Diagram]

• Deleting 7

FR-139: Deleting Leaves

[Diagram]

• Deleting 7

FR-140: Deleting Leaves

• If leaf contains 1 key
  - Cannot remove key without making leaf empty
  - Try to steal extra key from sibling

FR-141: Deleting Leaves

[Diagram]

• Steal key from sibling through parent
FR-142: **Deleting Leaves**

- Steal key from sibling through parent

FR-143: **Deleting Leaves**

- If leaf contains 1 key, and no sibling contains extra keys
  - Cannot remove key without making leaf empty
  - Cannot steal a key from a sibling
  - Merge with sibling
    - split in reverse

FR-144: **Merging Nodes**

- Removing the 4

FR-145: **Merging Nodes**

- Removing the 4
  - Combine 5, 7 into one node

FR-146: **Deleting Interior Keys**
• How can we delete keys from non-leaf nodes?
  • Replace key with smallest element subtree to right of key
  • Recursively delete smallest element from subtree to right of key
• (can also use largest element in subtree to left of key)

FR-147: **Deleting Interior Keys**

```
          4
         / \  /  \
        2   7
       / \ / \  / \  \
      1  3 5 6 8 9
```

• Deleting the 4

FR-148: **Deleting Interior Keys**

```
          4
         /  \
        2   7
       /  /  \
      1 3 5 6 8 9
```

• Deleting the 4
  • Replace 4 with smallest element in tree to right of 4

FR-149: **Deleting Interior Keys**

```
          5
         /  \
        2   7
       /  /  \
      1 3 6 8 9
```

FR-150: **Deleting Interior Keys**
• Deleting the 5

FR-151: Deleting Interior Keys

• Deleting the 5
  • Replace the 5 with the smallest element in tree to right of 5

FR-152: Deleting Interior Keys

• Deleting the 5
  • Replace the 5 with the smallest element in tree to right of 5
  • Node with two few keys

FR-153: Deleting Interior Keys
- Node with two few keys
- Steal a key from a sibling

**FR-154: Deleting Interior Keys**

```
      6
     /\  \\
    2 8
   /\  /\  \\
  1 3 7 9
```

**FR-155: Deleting Interior Keys**

```
      6 10
     /\  /\  \\
    2 8 11
   /\  /\  /\  \\
  1 3 7 9 12 13
```

- Removing the 6

**FR-156: Deleting Interior Keys**

```
      6 10
     /\  /\  \\
    2 8 11
   /\  /\  /\  \\
  1 3 7 9 12 13
```

- Removing the 6
  - Replace the 6 with the smallest element in the tree to the right of the 6

**FR-157: Deleting Interior Keys**

```
      7 10
     /\  /\  \\
    2 8 11
   /\  /\  /\  \\
  1 3 9 12 13
```
- Node with too few keys
  - Can't steal key from sibling
  - Merge with sibling

FR-158: **Deleting Interior Keys**

![Diagram of a 2-3 tree with nodes labeled 1 to 13 and keys 7, 10, 2, 11, 8, 9, 12, 13.]

- Node with too few keys
  - Can't steal key from sibling
  - Merge with sibling
  - (arbitrarily pick right sibling to merge with)

FR-159: **Deleting Interior Keys**

![Diagram of a 2-3 tree with nodes labeled 1 to 13 and keys 7, 2, 10, 11, 1, 3, 8, 9, 12, 13.]

FR-160: **Generalizing 2-3 Trees**

- In 2-3 Trees:
  - Each node has 1 or 2 keys
  - Each interior node has 2 or 3 children
- We can generalize 2-3 trees to allow more keys / node

FR-161: **B-Trees**

- A B-Tree of maximum degree k:
  - All interior nodes have \([k/2] \ldots k\) children
  - All nodes have \([k/2] - 1 \ldots k - 1\) keys
- 2-3 Tree is a B-Tree of maximum degree 3
FR-162: **B-Trees**

- B-Tree with maximum degree 5
  - Interior nodes have 3 – 5 children
  - All nodes have 2-4 keys

FR-163: **Connected Components**

- Subgraph (subset of the vertices) that is strongly connected.

FR-164: **Connected Components**

- Subgraph (subset of the vertices) that is strongly connected.

FR-165: **Connected Components**

- Subgraph (subset of the vertices) that is strongly connected.
FR-166: **Connected Components**

- Subgraph (subset of the vertices) that is strongly connected.

![Diagram of connected components](image)

FR-167: **DFS Revisited**

- We can keep track of the order in which we visit the elements in a Depth-First Search.
- For any vertex \( v \) in a DFS:
  - \( d[v] = \text{Discovery time} \) – when the vertex is first visited
  - \( f[v] = \text{Finishing time} \) – when we have finished with a vertex (and all of its children)

![Diagram of DFS example](image)

```java
class Edge {
    public int neighbor;
    public int next;
}

void DFS(Edge G[], int vertex, boolean Visited[], int d[], int f[]) {
    Edge tmp;
    Visited[vertex] = true;
    d[vertex] = time++;
    for (tmp = G[vertex]; tmp != null; tmp = tmp.next) {
        if (!Visited[tmp.neighbor]) {
            DFS(G, tmp.neighbor, Visited);
        }
        f[vertex] = time++;
    }
}
```

FR-168: **DFS Revisited**

![Diagram of DFS example](image)

FR-169: **DFS Example**

![Diagram of DFS example](image)

FR-170: **DFS Example**
FR-171: DFS Example

FR-172: DFS Example
FR-173: DFS Example

FR-174: DFS Example
FR-175: DFS Example

FR-176: DFS Example
FR-177: DFS Example

FR-178: DFS Example
FR-179: DFS Example

FR-180: DFS Example
FR-181: DFS Example

d 1  d 3  d  d  d  
f 10 f 8 f  f  f  

1  3  5  7

2  4  6  8

d 2  d 4  d 5  d 
f 9 f 7 f 6 f  

FR-182: DFS Example

d 1  d 3  d 11  d  
f 10 f 8 f  f  f  

1  3  5  7

2  4  6  8

d 2  d 4  d 5  d  
f 9 f 7 f 6 f  

FR-182: DFS Example
FR-183: DFS Example

FR-184: DFS Example
FR-185: DFS Example

```
   d 1   d 3   d 11  d 12
  f 10  f 8   f     f 13
```

```
1 -> 3

2

4

5 -> 7

6

8
```

```
   d 2   d 4   d 5   d 14
  f 9   f 7   f 6   f 15
```

FR-186: DFS Example

```
   d 1   d 3   d 11  d 12
  f 10  f 8   f 16  f 13
```

```
1 -> 3

2

4

5 -> 7

6

8
```

```
   d 2   d 4   d 5   d 14
  f 9   f 7   f 6   f 15
```

FR-187: Using d[] & f[]

- Given two vertices $v_1$ and $v_2$, what do we know if $f[v_2] < f[v_1]$?
  - Either:
    - Path from $v_1$ to $v_2$
      - Start from $v_1$
      - Eventually visit $v_2$
      - Finish $v_2$
      - Finish $v_1$

FR-188: Using d[] & f[]

- Given two vertices $v_1$ and $v_2$, what do we know if $f[v_2] < f[v_1]$?
  - Either:
Path from $v_1$ to $v_2$
No path from $v_2$ to $v_1$
  • Start from $v_2$
  • Eventually finish $v_2$
  • Start from $v_1$
  • Eventually finish $v_1$

FR-189: Using $d[]$ & $f[]$

• If $f[v_2] < f[v_1]$:
  • Either a path from $v_1$ to $v_2$, or no path from $v_2$ to $v_1$
  • If there is a path from $v_2$ to $v_1$, then there must be a path from $v_1$ to $v_2$
• $f[v_2] < f[v_1]$ and a path from $v_2$ to $v_1 \Rightarrow v_1$ and $v_2$ are in the same connected component

FR-190: Connected Components

• Run DFS on $G$, calculating $f[]$ times
• Compute $G^T$
• Run DFS on $G^T$ – examining nodes in inverse order of finishing times from first DFS
• Any nodes that are in the same DFS search tree in $G^T$ must be in the same connected component

FR-191: Dynamic Programming

• Simple, recursive solution to a problem
• Naive solution recalculates same value many times
• Leads to exponential running time

FR-192: Dynamic Programming

• Recalculating values can lead to unacceptable run times
  • Even if the total number of values that needs to be calculated is small
• Solution: Don’t recalculate values
  • Calculate each value once
  • Store results in a table
  • Use the table to calculate larger results

FR-193: Faster Fibonacci

```java
int Fibonacci(int n) {
    int[] FIB = new int[n+1];
    FIB[0] = 1;
    FIB[1] = 1;
    // Your implementation here...
}
```
for (i=2; i<=n; i++)
    FIB[i] = FIB[i-1] + FIB[i-2];
return FIB[n];
}

FR-194: **Dynamic Programming**

- To create a dynamic programming solution to a problem:
  - Create a simple recursive solution (that may require a large number of repeat calculations)
  - Design a table to hold partial results
  - Fill the table such that whenever a partial result is needed, it is already in the table

FR-195: **Memoization**

- Can be difficult to determine order to fill the table
- We can use a table together with recursive solution
  - Initialize table with sentinel value
  - In recursive function:
    - Check table – if entry is there, use it
    - Otherwise, call function recursively
      - Set appropriate table value
      - return table value

FR-196: **Fibonacci Memoized**

```c
int Fibonacci(int n) {
    if (n == 0)
        return 1;
    if (n == 1)
        return 1;

    if (T[n] == -1)
        T[n] = Fibonacci(n-1) + Fibonacci(n-2);

    return T[n];
}
```