CPSC 331: DATA STRUCTURES, ALGORITHMS, AND THEIR ANALYSIS

GO UEZONO

April 13, 2024

CONTENTS

1 Algorithmic Analysis 2
- 1.1 Mathematical Induction 3
- 1.2 Loop Invariants 4
- 1.3 Bound Functions 4

2 Elementary Data Structures 5
- 2.1 Lists 5
- 2.2 Linked Lists 6
- 2.3 Stacks 9
- 2.4 Queues 11

3 Data Structures 12
- 3.1 Binary Search Trees 12
- 3.2 Red and Black Trees 16
- 3.3 Heaps 23
- 3.4 Hash Table 29
- 3.5 Graphs 36
- 3.6 Dijkstra's Algorithm 38

4 Sorting Algorithms 42
- 4.1 Bubble Sort 42
- 4.2 Selection Sort 45
- 4.3 Insertion Sort 47
- 4.4 Heap Sort 50

5 Advanced Sorting Algorithms 52
- 5.1 Merge Sort 52
- 5.2 Quick Sort 53

6 Searching 56
- 6.1 Linear Search 56
- 6.2 Binary Search 58

7 Graph Traversal 60
- 7.1 Depth-First Search 60
- 7.2 Breadth-First Search 61

1 ALGORITHMIC ANALYSIS

Log Laws

- If base is 2, then it is not written
- $\log_b(a) = c$ if $a = b^c$
- $\log_y(x) = z$ then $y^z = x$
- $\log(n)^c = \log^c n$
- $\log(\log(n)) = \log\log n$

Common Derivatives

- $\frac{d}{dx}\ln(x) = \frac{1}{x}$
- $\frac{d}{dx}\log(x) = \frac{1}{x\ln(2)}$
- $\frac{d}{dx}\log_y(x) = \frac{1}{x\ln(y)}$

L'Hopital's Rule

- $$\lim_{x\to\infty}\frac{f(x)}{g(x)} = \lim_{x\to\infty}\frac{f'(x)}{g'(x)}$$
- **Limit Test**
  - If,
    $$\lim_{x\to\infty}\frac{f(x)}{g(x)}$$
    exists and is a real and non-negative number, then f is $O(g(x))$
  - **Example**: $4x^2 + 2 \in O(x^2)$
    $$\lim_{x\to\infty}\frac{4x+2}{x^2} = \lim_{x\to\infty} 4 + \frac{2}{x^2} = 4$$
    Thus, $4x^2 + 2$ is $O(x^2)$
  - **Example**: $\log(x) \in O(x)$
    $$\lim_{x\to\infty}\frac{\log(x)}{x}$$
    Now, by L'Hopital's Rule,
    $$\lim_{x\to\infty}\frac{\frac{1}{x\ln(2)}}{1} = \lim_{x\to\infty}\frac{1}{x\ln(2)} = 0$$
    Thus, $\log(x)$ is $O(x)$

Big-Oh Notation

- Let n be a constant that grows to ∞
  - If $g(n)$ is $O(f(n))$, then $g(n)$ is not worse than $f(n)$
  - If $g(n)$ is $\Omega(f(n))$, then $g(n)$ is not better than $f(n)$
  - If $g(n)$ is $\Theta(f(n))$, then $g(n)$ is not better or worse than $f(n)$

Function	Name
$O(1)$	Constant
$O(\log(n))$	Log
$O(n)$	Linear
$O(n^2)$	Quadratic
$O(n^k)$	Polynomial
$O(a^n), a > 1$	Exponential

1.1 Mathematical Induction

Standard Form

- $P(n)$ is property that is defined $\forall n$
- α a fixed int
- Suppose:
  - $P(\alpha)$ is true
  - $\forall k \geqslant \alpha$, if $P(k)$ is true then $P(k+1)$ is true
- Then, $P(n)$ is true for every int $n \geqslant \alpha$
- Consists of 2 steps:
  - *Basis/Base case*: Show that $P(\alpha)$ is true (usually $\alpha = 0$)
  - *Inductive Step*: k be an int such that $k \geqslant \alpha$, hence
    - **Inductive Hypothesis (assume)**: $P(k)$ is true
    - **Inductive Claim (prove)**: $P(k+1)$ is true
  - Conclude $P(n)$ is true for every int $n \geqslant \alpha$

Strong Form

- $P(n)$ is property that is defined $\forall n$
- α, β fixed ints such that $\alpha \leqslant \beta$
- Suppose:
  - $P(\alpha), P(\alpha+1), P(\alpha+2), ..., P(\beta)$ all true
  - $\forall k \geqslant \beta$, if $P(i)$ is true for $\forall i$ such that $\alpha \leqslant i \leqslant k$, then $P(k+1)$ is true
- Then, $P(n)$ is true for every int $n \geqslant \alpha$
- Consists of 2 steps but with an additional:
  - Choice of Breakpoint: Choosing β such that $\beta \geqslant \alpha$
  - *Basis/Base case*: Show that $P(\alpha), P(\alpha+1), P(\alpha+2), ..., P(\beta)$ are all true
  - *Inductive Step*: k be an int such that $k \geqslant \beta$, hence
    - **Inductive Hypothesis (assume)**: $P(i)$ is true for $\forall i$ such that $\alpha \leqslant i \leqslant k$
    - **Inductive Claim (prove)**: $P(k+1)$ is true
  - Conclude $P(n)$ is true for every int $n \geqslant \alpha$

M.I Techniques

- Manipulation
- Simplification

Conclusion

- To determine which form of MI to use, if the IH in **standard form** does not give enough information to prove $P(n+1)$ is true, **strong form** is ideal

1.2 Loop Invariants

- Mainly used for **Iterative** algorithms
- A property that must satisfy 3 conditions:
  - **Initialization**: Loop Invariant is true before the loop
  - **Maintenance**: Loop Invariant is true after each iteration
  - **Termination**: Loop Invariant should state something to help understand algorithm when loop terminates

1.3 Bound Functions

- Mainly used for **recursive** algorithms
- Used to prove termination of int return type algorithms.
- Let f be the bound function for some recursive algorithm $A(p)$ where p is a list of parameters. Then for an integer, k, the following hold:
  1. $f(p) \leqslant k$ for the base case values.
  2. $f(p) > k$ for the recursive case values.
  3. For each recursive iteration of $A(q)$, with $q < p$, then $f(q) < f(p)$. That is, the value of A for each recursive iteration converges towards a base case in order to guarantee termination
- **Example**:

```
    int fact(int n) {
       if (n <= 1) {
          return 1;
       }
       return n * fact(n - 1);
    }
```

- Let $f(n) = n, n \geqslant 0$
  - $f(0) \leqslant k$ for the base case.
  - $f(p) > k$ for the recursive case.
  - $q < p \to f(q) < f(p)$.

Proof. – Pick k = 0

- Then $f(0) \leqslant k \leqslant 0$ and property 1 is satisfied
- $f(p)$ is guaranteed to be greater than $k = 0$ for the recursive case by definition
- For each recursive iteration of $\mathrm{fact}(n - m)$ with $n < m$ it is the case that $f(n - m) < f(n)$
- Thus, $\mathrm{fact}(n), n \geqslant 0$ terminates

□

2 ELEMENTARY DATA STRUCTURES

2.1 Lists

2.1.1 *Algorithm*

```
// Array implementation Unsorted
void add(T item) {
   if (!isFull()) {
      list[size] = item;
      size++;
   } else {
      throw exception
   }
}
int contains(T item) {
   if (!isEmpty()) {
      int i = 0;
      while (i < size) {
         if (item.compareTo(list[i]) == 0) {
            return i;
         }
      }
   }
   return -1;
}
void remove(T item) {
   if (!isEmpty()) {
      int i = 0;
      while (i < size) {
         if (item.compareTo(list[i] == 0)) {
            list[i] = list[size - 1];
            size--;
            return;
         }
         i++;
      }
   } else {
      throw exception
   }
}
```

2.1.2 *Time Complexity*

Unsorted List

Operation	**Complexity**
add()	$O(1)$
contains()	$O(n)$
remove()	$O(n)$

2.1.3 *Correctness*

2.2 Linked Lists

2.2.1 *Algorithm*

```
// Singly
public void prepend(T item) {
   if (item == null) return;
   if (this.head == null) {
      this.head = new Node<T>(item, null);
   } else {
      Node<T> tmp = this.head;
      this.head = new Node<T>(item, tmp);
   }
   this.size++;
}
public void append(T item) {
   if (item == null) return;
   if (this.head == null) {
      this.head = new Node<T>(item, null);
   } else {
      Node<T> tmp = this.head;

      while (tmp.next != null) {
         tmp = tmp.next;
      }
      tmp.next = new Node<T>(item, null);
   }
   this.size++;
}
public Node<T> deleteFirst() {
   if (this.head == null) return null;
   Node<T> tmp = this.head;
   this.head = this.head.next;
   this.size--;
   return tmp;
}
public Node<T> deleteLast() {
   if (this.head == null) return null;
   Node<T> tmp = this.head;
   Node<T> prev = null;
   while (tmp.next != null) {
      prev = tmp;
      tmp = tmp.next;
   }
   // 1 item case
   if (prev == null) {
      // Only one node in the linked list
      this.head = null;
   } else {
      prev.next = null;
   }
   this.size--;
   return tmp;
}
// Doubly
public void add(T element) {
   Node newNode = new Node(element);
   if (head == null) {
      head = newNode;
```

```
         tail = newNode;
      } else {
         tail.next = newNode;
         newNode.prev = tail;
         tail = newNode;
      }
      size++;
   }
   public int contains(T element) {
      Node temp = head;
      int index = 0;
      while (temp != null) {
         if (temp.data.equals(element)) {
            return index;
         }
         temp = temp.next;
         index++;
      }
      return -1;
   }
   public void remove(T element) {
      Node temp = head;
      while (temp != null) {
         if (temp.data.equals(element)) {
            if (temp == head) {
               head = temp.next;
            }
            if (temp == tail) {
               tail = temp.prev;
            }
            if (temp.prev != null) {
               temp.prev.next = temp.next;
            }
            if (temp.next != null) {
               temp.next.prev = temp.prev;
            }
            size--;
            return;
         }
         temp = temp.next;
      }
   }
```

2.2.2 *Time Complexity*

Singly LL

Operation	Complexity
prepend()	$O(1)$
append()	$O(n)$
deleteFirst()	$O(1)$
deleteLast()	$O(n)$

Doubly LL

Operation	Complexity
prepend()	$O(1)$
append()	$O(1)$
deleteFirst()	$O(1)$
deleteLast()	$O(1)$

2.2.3 *Correctness*

Loop Invariant

- ***Initialization***
  - Before entering the loop, tmp is initialised to the head of the list and prev is null since there is no node before it (prev would be the last element in the list initially for circular list)
- ***Maintenance***
  - During each iteration of the loop, tmp is updated to *tmp.next*, moving to the next node in the list, and prev is updated to the previous value of tmp
  - This ensures that at the start of each iteration, tmp points to the current node, and prev points to the node preceding tmp
- ***Termination***
  - The loop terminates when *tmp.next* is null (or say tmp. next is head for a circular list) indicating that tmp is the last node in the list. At this point, prev points to the second to last node or null if only one element exists in the list

2.3 Stacks

2.3.1 *Algorithm*

```
// Array Implementation
public void push(T item) {
   if (!isFull) {
      topIndex++;
      stack[topIndex] = item;
   } else {
      throw new OverflowException("Cannot push onto full stack");
   }
}
public void pop() {
   if (!isEmpty) {
      T tmp = stack[topIndex];
      stack[topIndex] = null;
      topIndex--;
      return tmp;
   } else {
      throw exception
   }
}
public T peek() {
   if (!isEmpty()) {
      return stack[topIndex];
   } else {
      throw exception
   }
}

// LinkedList Implementation
public void push(T item) {
   Node<T> nn = new Node<T>();
   nn.value = item;
   nn.next = top;
   top = nn;
}
public T pop() {
   if (!isEmpty()) {
      T tmp = top.value;
      top = top.next;
      return tmp;
   } else {
      throw exception
   }
}
public T peek() {
   if (!isEmpty()) {
      return top.value;
   } else {
      throw exception
   }
}
```

2.3.2 *Time Complexity*

ADT Array Stack

Operation	**Complexity**
isFull()	$O(1)$
isEmpty()	$O(1)$
push()	$O(1)$
peek()	$O(1)$
pop()	$O(1)$

ADT Linked Stack

Operation	**Complexity**
isFull()	$O(1)$
isEmpty()	$O(1)$
push()	$O(1)$
peek()	$O(1)$
pop()	$O(1)$

2.4 Queues

2.4.1 *Algorithm*

```
// Array Implementation
public void enqueue(T item) {
   rear = (rear + 1) % queue.length;
   queue[rear] = item;
   size++;
}
public T dequeue() {
   T tmp = queue[front];
   queue[front] = null;
   front = (front + 1) % queue.length;
   size--;
   return tmp;
}

// Linked Implementation
public void enqueue(T item) {
   Node nn = new Node<T>();
   nn.value = item;
   nn.next = rear;
   rear = nn;
}
public T dequeue() {
   T tmp = front.value;
   if (front == rear) {
      front == null;
      rear == null;
   } else {
      Node<T> tmN = rear;
      while (tmN.next != front) tmN = tmN.next;
      front = tmN;
      front.next = null;
   }
   return tmp;
}
```

2.4.2 *Time Complexity*

ADT Array Queue

Operation	**Complexity**
isFull()	$O(1)$
isEmpty()	$O(1)$
enqueue()	$O(1)$
dequeue()	$O(1)$

ADT Linked Queue

Operation	**Complexity**
isFull()	$O(1)$
isEmpty()	$O(1)$
enqueue()	$O(1)$
dequeue()	$O(1)$

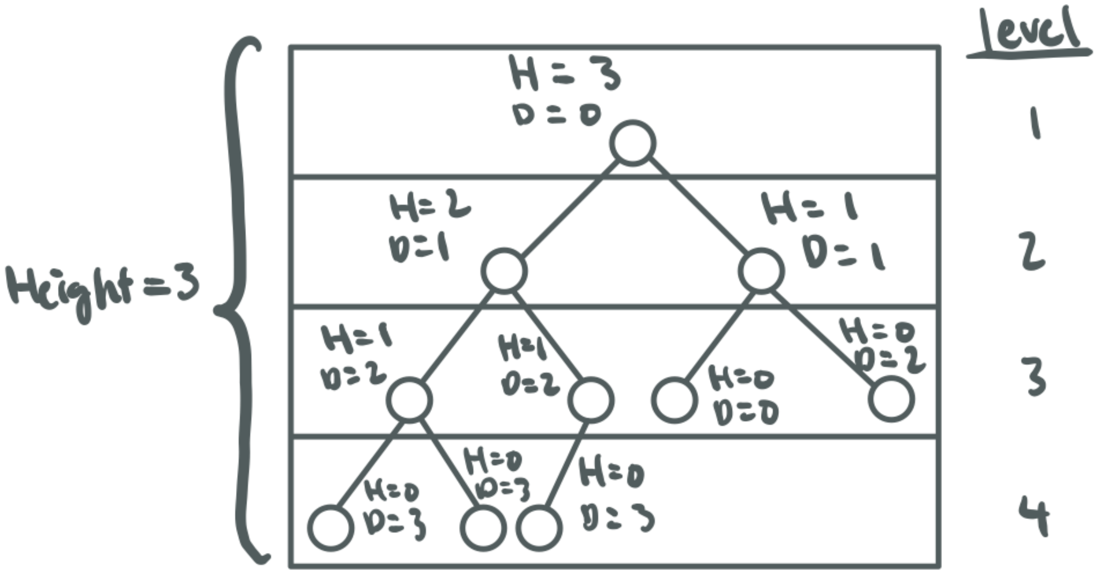
3 DATA STRUCTURES

3.1 Binary Search Trees

3.1.1 *Trees*

Tree Properties

- **Path**
  - Series of nodes to a leaf
  - Length of less than the number of nodes encountered, length $m - 1$
- **Degree** of *n*
  - Number of children of any Node *n*
- **Depth** of *n*
  - Length of the simplest path from the root to *n*
- **Level**
  - Consists of all nodes at the same depth
  - Equal to depth if the level starts from depth 0
  - Equal to depth + 1 if level started from 1
  - Equal to height - 1
- **Height** of *n*
  - Length of longest path from root to a leaf

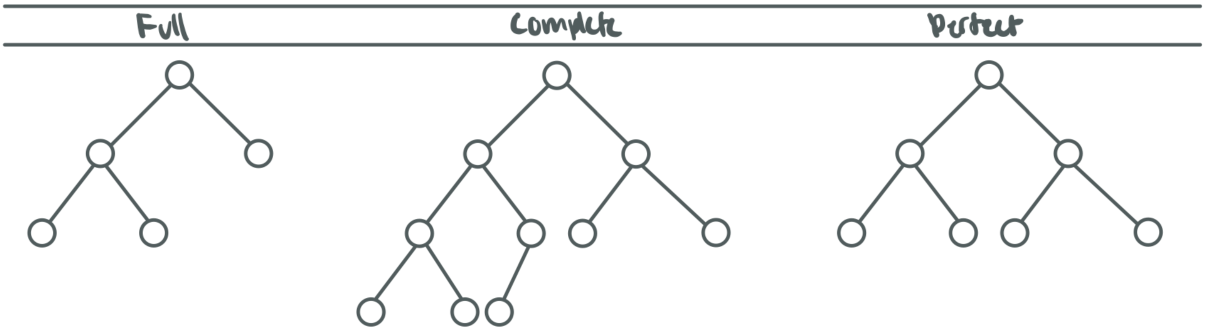


3.1.2 *Properties*

BST Properties

- Ordered tree where each node can have up to 2 children

- Called left and right children by convention
- **Full**
  - Every node has either exactly 2 children or no children
- **Complete**
  - Tree is filled at all levels except the last level
- **Perfect**
  - It is full and all leaves are at the same level



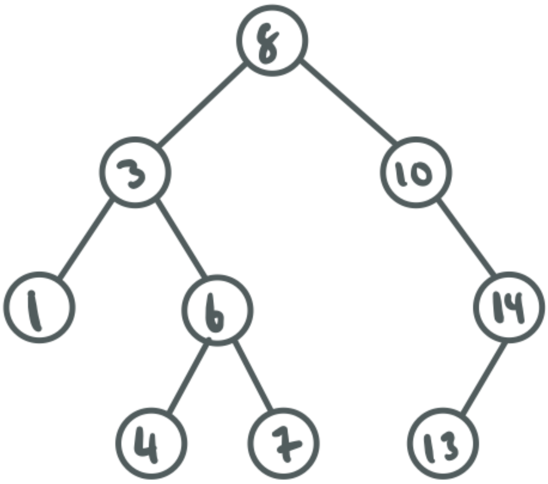
Prove that the Height of a Complete Tree is $O(\log(n))$

- A complete tree of height h is perfect up to the height $h-1$
- That is, h is $O(\log(n))$
- **Claim**: The height, h, of a complete tree with $n \geqslant 1$ nodes is $O(\log(n))$

  Proof. – Let $P(n)$ be the assertion that height, h, of a complete tree with n nodes is $O(\log(n))$
  - **Basis**: $P(1)$ A tree with 1 node, the root, has a height of $O(\log(1))$
    - This evaluates to $O(0)$ or simply a constant
    - This is true because a single node tree has a height of 0
  - **Inductive Step**: Assume $P(k)$ holds for all k such that $1 \leqslant k < n$
    - That is, any complete tree with k nodes has a height, $h = O(\log(k))$
    - Now, consider a complete tree with *j* nodes. This tree can be divided into 2 parts:
      - A sub tree with $\frac{j}{2}$ nodes
      - Another tree with $\frac{j}{2} - 1$ nodes
    - By the **inductive hypothesis**, the height of the sub tree with $\frac{j}{2}$ nodes is $O(\log(\frac{j}{2})) = O(\log(j))$
    - The heigh of the subtree with $\frac{j}{2} - 1$ nodes is $O(\log(\frac{j}{2} - 1)) = O(\log(j))$
    - The height of the complete tree with j nodes is one more than the height of its taller sub tree
    - Hence, he height of the complete tree is at most $O(\log(j))$
  - Since the **basis** and the **inductive step** hold, the claim is established by mathematical induction

□

BST Traversal



- **In-Order**: 1, 3, 4, 6, 7, 10, 13, 14
  - left subtree → root → right subtree
- **Pre-Order**: 8, 3, 1, 6, 4, 7, 10, 14, 13
  - root → left subtree → right subtree
- **Post-Order**: 1, 4, 7, 6, 3, 13, 14, 10, 8
  - left subtree → right subtree → root
- **Predecessor**: The next node in an in-order traversal with the largest value that is smaller than the node itself
  - e.g. 4 is the predecessor of 6, 1 is the predecessor of 3
  - Comes immediately before in an in-order traversal
- **Successor**: The previous node in an in-order traversal with the smallest value that is greater than the node itself
  - 7 is the successor of 6, 6 is the successor of 4

3.1.3 *Algorithm*

```
// Search
contains(T item, Node root) {
   if (root == null) {
      return false;
   } else if (item.compareTo(root.value) < 0) {
      return recContains(item, root.left);
   } else if (item.compareTo(root.value) > 0) {
      return recContains(item, root.right);
   } else {
      return true;
   }
}
// Insert
recAdd(T item, Node root) {
   if (root == null) {
      root = new Node<T>();
      root.value = item;
```

```
        } else if (item.compareTo(root.value) < 0) {
            root.left = recAdd(item, root.left);
        } else {
            root.right = recAdd(item, root.right);
        }
        return root;
    }
    // Delete
    recRemove(T item) {
        if (root == null) {
            return null;
        } else if (item.compareTo(root.value) < 0) {
            root.left = recRemove(item, root.left);
        } else if (item.compareTo(root.value) > 0) {
            root.right = recRemove(item, root.right);
        } else {
            root = removeNode(root);
        }
        return root;
    }
    removeNode(Node<T> root) {
        T tmp;
        if (root.left == null) {
            return root.right;
        } else if (root.right == null) {
            return root.left;
        } else {
            tmp = predecessor(root.left);
            root.value = tmp;
            root.left = recRemove(tmp, root.left);
            return root;
        }
    }
```

3.1.4 *Time Complexity*

BST relative to Height

Operation	Complexity
search()	$O(h)$
insert()	$O(h)$
delete()	$O(h)$

BST

Operation	Complexity
search()	$O(n)$
insert()	$O(n)$
deleted()	$O(n)$

Balanced BST

Operation	Complexity
search()	$O(\log(n))$
insert()	$O(\log(n))$
deleted()	$O(\log(n))$

3.2 Red and Black Trees

3.2.1 *Properties*

A **red-black tree** is a concrete implementation of a **self-balancing binary-search tree** (reference here) that automatically maintains balance. Giving each node their respective color ensures that no path is more than twice as long as any other, thus is able to maintain approximate balance.

1. Every node is red/black
2. Root must be black
3. Leaves (*null*) are black
   - *null* vertices contain no values, while other (interior) do
4. If a node is red, then both its children are black
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes

The following **lemma** shows why red-black trees make good search trees:

Lemma 1. *A red-black tree with* n *internal nodes has height at most* $2\log(n+1)$

Proof. Start by showing subtree rooted at any node x contains at least a $2^{bh(x)} - 1$ internal nodes. We prove this by **mathematical induction** on the height of x.

CLAIM: If height of $x = 0$, then the leaf must be *T.null*, and the subtree rooted at x contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

Inductive step:

- Consider a node x that has positive height and is an internal node with two children.
- Each *child* has a black-height of either $bh(x)$ or $bh(x) - 1$ (depending on whether it is red or black respectively).
- Since height of a *child* of x is less than the height of x itself, we can apply the **I.H** to conlude that:
  - Each child has at least $2^{bh(x)-1} - 1$ internal nodes.

Thus, subtree rooted at x contains at least

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$$

internal nodes, which proves the claim.

To complete the proof, let h be the height of thr tree. According to property 4 (reference above Properties), at least half the nodes from the root to a leaf (not including the root) must be black.

Consequently, the bh of the root must be at least $h/2$; thus,

$$n \geqslant 2^{h/2} - 1$$

Moving 1 to the left side and taking log on both sides yields:

$$\log(n+1) \geqslant h/2$$

or

$$h \leqslant 2\log(n+1)$$

□

3.2.2 *Rotational Properties*

Search operations *TREE-INSERT* and *TREE-DELETE* take $O(\log n)$ time. Since modifications are done to the tree, we must change the color of some of the nodes.

Definition (Rotation). Local operation that preserves the binary-tree property.

- **Left Rotation:** assume that its right child y is not *null*
- **Right Rotation:** assume that its left child y is not *null*
  - x can be any node on the tree whose respective child is not *null*
  - Left/Right rotations "pivots" around the link from x to y
  - Makes y the new root, x as y's left(right) child, y's left(right) child as x's right(left) child
- Both L/R rotates run in $O(1)$ time
- Only pointers are changed, all attributes in a node remain the same

Algorithm 1: Left-Rotate(T, x)

```
x = y.right ;                                      // set y
x.right = y.left ;      // Turn y's left subtree into x's right subtree
if y.left ≠ T.null then
    y.left.p = x;
y.p = x.p;
if x.p == T.null then
    T.root = y;
else if x == x.p.left then
    x.p.left = y;
else
    x.p.right = y;
y.left = x;
x.p = y;
```

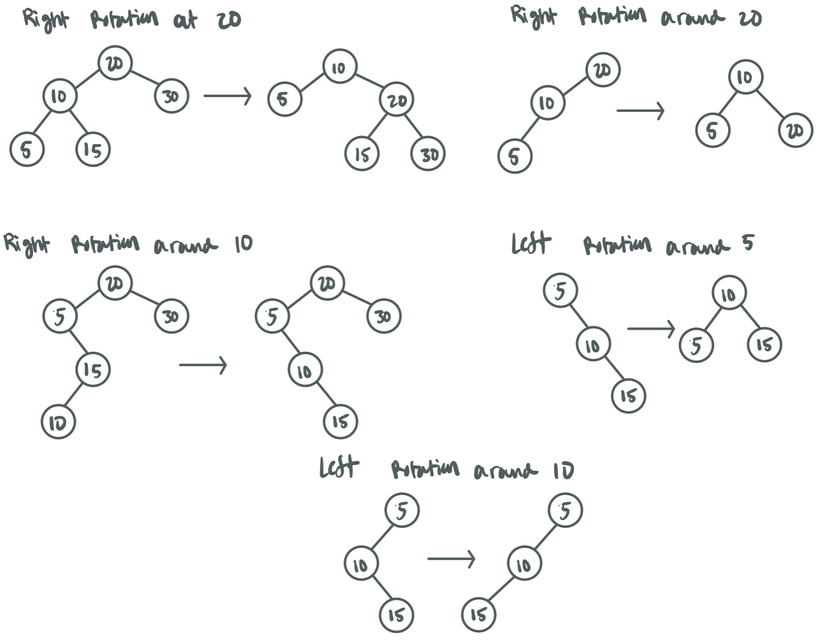


Figure 1: DB Rotations

3.2.3 *Insertion*

General Rules:

1. If tree is empty, insert node as root and color B
2. If tree is **not** empty, insert new node as leaf and color R
3. If Parent *P* is B, exit
4. Else if *P* is R, check color of Uncle *U*
    - If *U* is B/null, Rotate and Recolor
    - If *U* is R, Recolor and check Grandparent *GP*
        - If *GP* is **not** root, Recolor and Recheck again

- If 3 nodes (e.g. A, B, C) are straight, coloring will involve the *GP* and *P* (that is, A and B) but **not** Node (C)
- For a path from GP→P→N:
    - If it is left→right or right→left, follow the indicated rotations
    - If nodes are straight with right→right, left rotation
    - If nodes are straight with left→left, right rotation

Inserting a node can be done in $O(\log n)$ time. Below is a pseudo-code that shows how insertion *RB-INSERT* works:

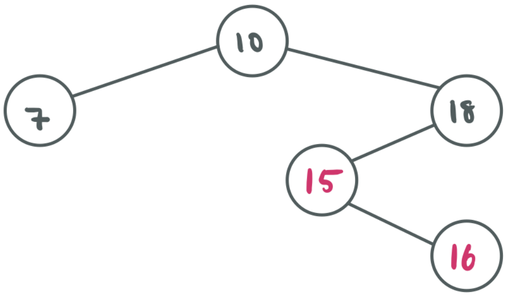
Algorithm 2: RB-INSERT(T, z)

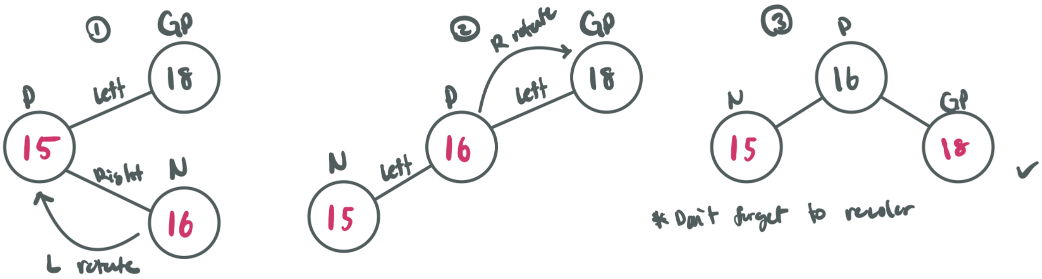
```
Data: $z$ node to insert,
$y = T.null$;
$x = T.root$;
while $x \neq T.null$ do
    $y = x$;
    if $z.key < x.key$ then
        $x = x.left$;
    else
        $x = x.right$;
    end
end
$z.p = y$;
if $y == T.null$ then
    $T.root = z$;
else if $z.key < y.key$ then
    $y.left = z$;
else
    $y.right = z$;
$z.left = T.null$;
$z.right = T.null$;
$z.color = T.RED$;
RB-INSERT$(T, z)$;
```

Insert: 10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70

Insert

1) 10
2) 18
3) 7
4) 15
   - recolor P, U
   - Check GP
5) 16
   - U is null
   ↳ Rotate, Recolor
   ↳ Path is L→R, do L then R rotate

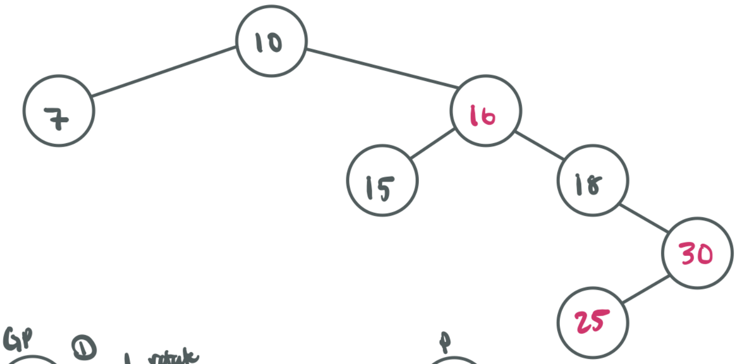


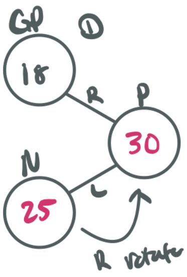


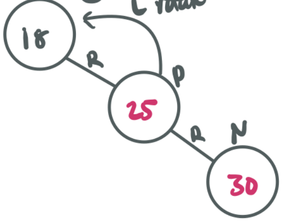
* Don't forget to recolor

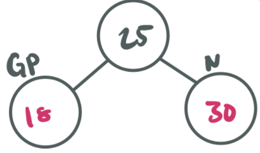
6) 30
   - U is R, recolor
   - GP ≠ root, recolor & recheck

7) 25
   - U = null
   ↳ rotate, recolor









* recolor!

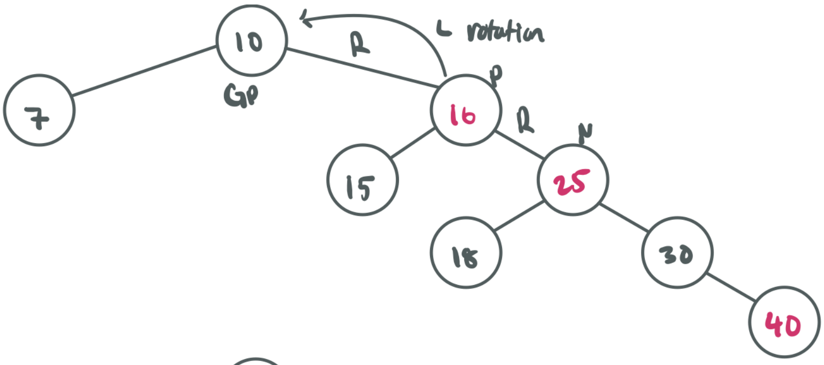
8) 40

- U is R, recolor
  - check GP
  - ↳ GP ≠ root, recolor recheck

25

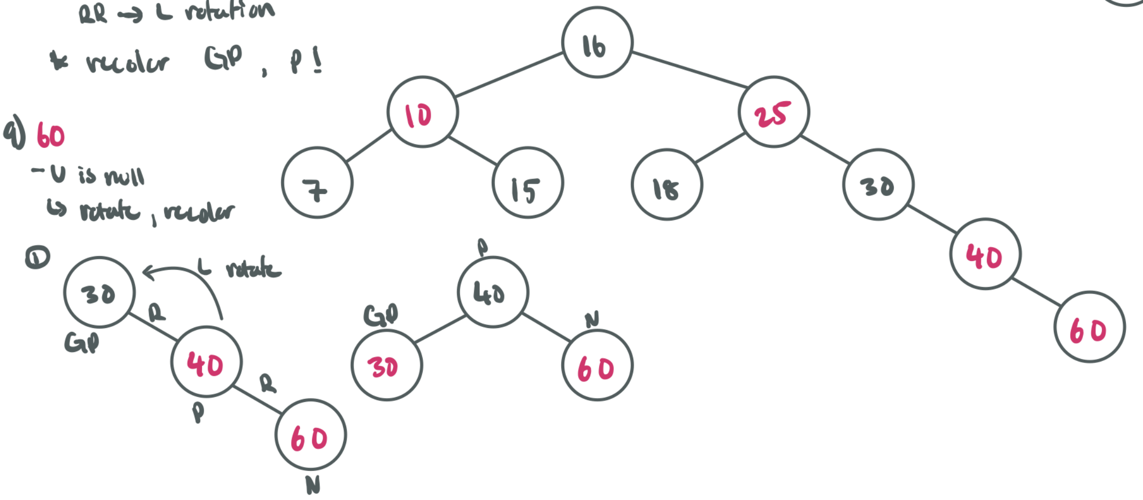
- P is R, U is B
  - ↳ rotate, recolor
  - RR → L rotation

* recolor GP, P!

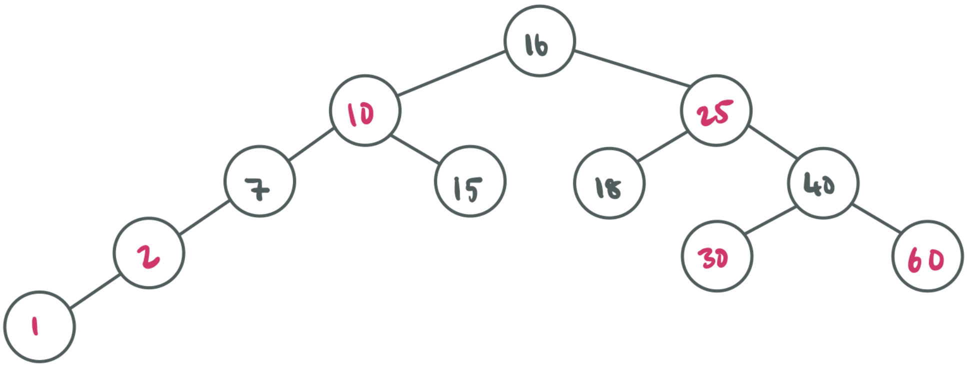


9) 60

- U is null
  - ↳ rotate, recolor

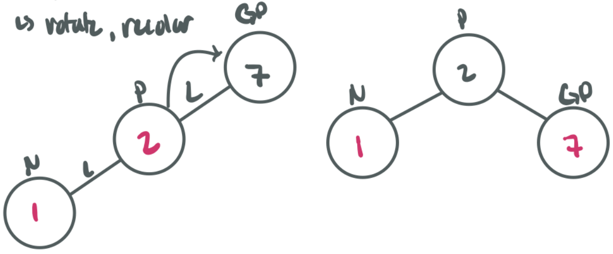


10) 2



11) 1

- U is null
  - ↳ rotate, recolor



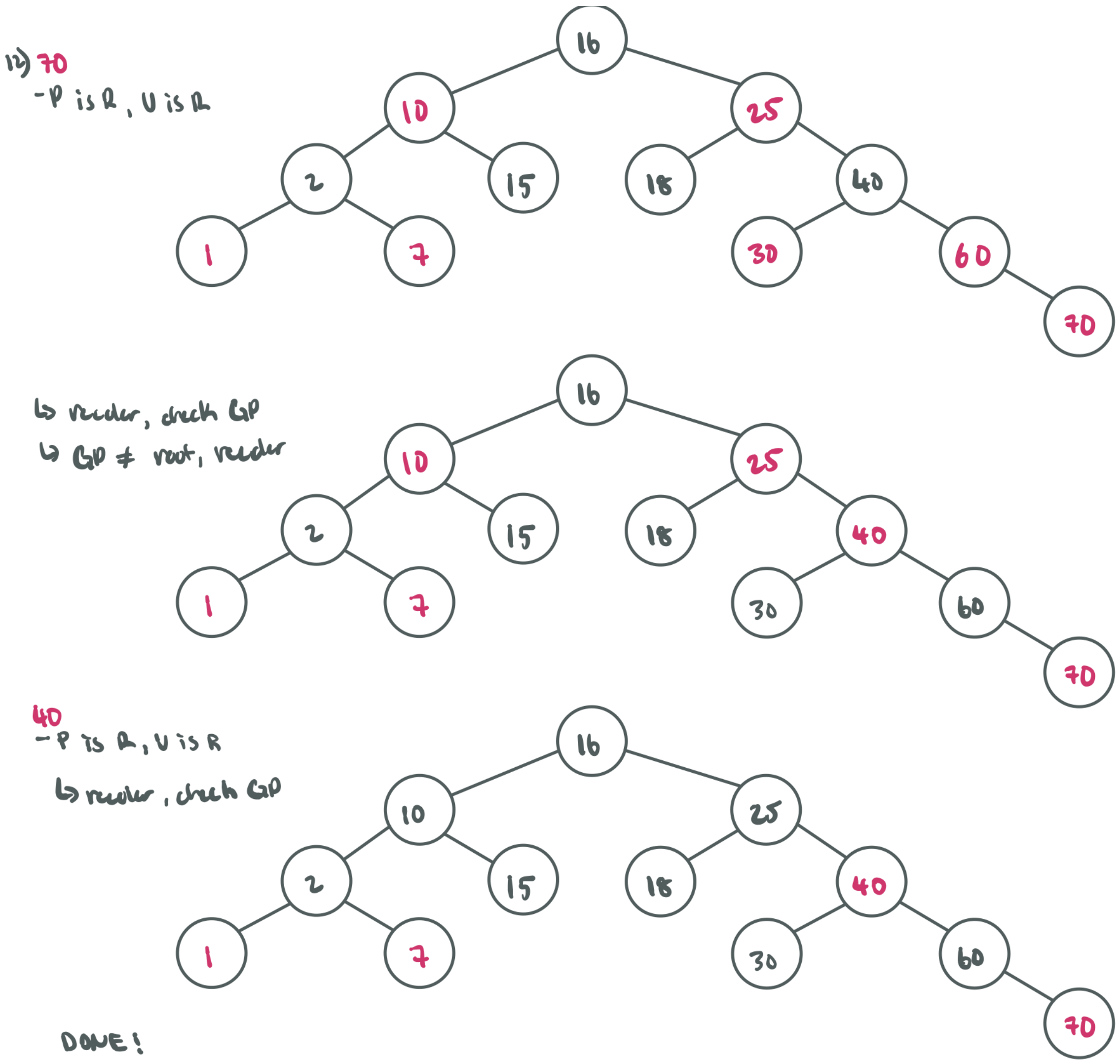


Figure 2: Example of RBT Insert

Correctness

- Add() function starts with the same method as the BST add(); thus resulting tree is still a BST
- Rotation functions do not change the BST property
- Prove that the rotations and coloring in fixAfterAdd() maintains the RBT Properties

Time Complexity

- Height of an RBT is $\leqslant 2\log(n+1)$, where n is the number of values (non-null vertices) in the tree
- Adding a node is $O(\log n)$ W.C

3.2.4 *Deletion*

6 Cases

1. If node is red *r*, just delete
2. If root is double black *db*, just remove *db*

3. If *db*'s sibling *s* is black *b* and both its children are *b*
   - Remove *db*
   - Add *b* to its Parent *p* (if *r* becomes *b*, else *db*)
   - Make *s r*
   - If *db* still exists, apply other cases
4. If *db s* is *r*
   - Swap colors of *p* and its *s*
   - Rotate *p* in *db* direction
   - Reapply necessary cases
5. *db*'s *s* is *b*, *s* child far from *db* is *b*, but near child to *db* is *r*
   - Swap color of *db*'s *s* and *s* child who is near *db*
   - Rotate *s* in opposite direction to *db*'s direction
   - Apply case 6
6. *db*'s *s* is *b*, far child is *r*
   - Swap color of *p* and *s*
   - Rotate *p* in *db*'s direction
   - Remove *db*
   - Change color of *r* child to *b*

Correctness

- Remove() function starts with the same method as the BST remove(); thus resulting tree is still a BST
- Rotation functions do not change the BST property
- Prove that the rotations and coloring in fixAfterRemove() maintains the RBT Properties

Time Complexity

- Height of an RBT is $\leqslant 2\log(n+1)$, where n is the number of values (non-null vertices) in the tree
- Removing a node is $O(\log n)$ W.C

3.3 Heaps

3.3.1 *Priority Queues*

- Priority queues are **NOT** FIFO
- These queues are interested in removing items (dequeue) with the **highest priority**
- Assume that higher priority value (HPV) entails higher priority
  - Not true in general (UNIX OS; smaller PV = higher priority)
- Similar operations as the standard Queue ADT:

```java
// Queue ADT
public interface QueueADT<T> {
   public void enqueue(T item);
   public T dequeue(); // different implementation
   public boolean isEmpty();
   public boolean isFull();
}
```

PRIORITY QUEUE IMPLEMENTATIONS

- **Lists**
  - **Sorted** list by PV with array implementation
    - *Ascending order*: Remove **last** item
    - *Dequeue*: $O(1)$, n^{th} element of the array to be removed
    - *Enqueue*: $O(n)$, needs to be sorted after enqueue
  - **Sorted** list by PV with linked-list implementation
    - *Descending order*: Remove **first** item
    - *Ascending order*: Circular list implementation
    - *Dequeue*: $O(1)$
    - *Enqueue*: $O(n)$
  - **Unsorted** list
    - *Dequeue*: $O(n)$
    - *Enqueue*: $O(1)$
- **BST**
  - *Dequeue*: $O(\log(n))$ average-case, $O(n)$ worst-case
  - *Enqueue*: $O(\log(n))$ average-case, $O(n)$ worst-case
- **Heaps**
  - *Dequeue*: $O(\log(n))$ worst-case
  - *Enqueue*: $O(\log(n))$ worst-case

3.3.2 *Properties*

A heap is a **complete** binary tree that satisfies the Heap Property:

- Each node of a tree corresponds to an element of an array
  - Always stored **contiguously**
    - If there are blanks, they are on the rightside of the array
    - Otherwise, no blanks inbetween indices
- It is of height h and contains n nodes

Proof. Height h □

HEAP PROPERTIES (Figure 3)

- **Min Heap Property**
  - Every node has a value $\leqslant$ than the value of its children
  - Root of any subtree has the *minimum* value in the subtree
- **Max Heap Property**
  - Every node has a value $\geqslant$ than the value of its children
  - Root of any subtree has the *maximum* value in the subtree
- **Child-to-Parent Relation**
  - Given a **child** a location *loc*, what is the index of **parent** *parent*?
  - If Child is right, index is **even**
    - $\mathtt{loc}_r = 2 \times \mathtt{parent} + 2$
  - If Child is left, index is **odd**
    - $\mathtt{loc}_l = 2 \times \mathtt{parent} + 1$
  - In either cases, $\mathtt{parent} = (\mathtt{loc} - 1) \div 2$

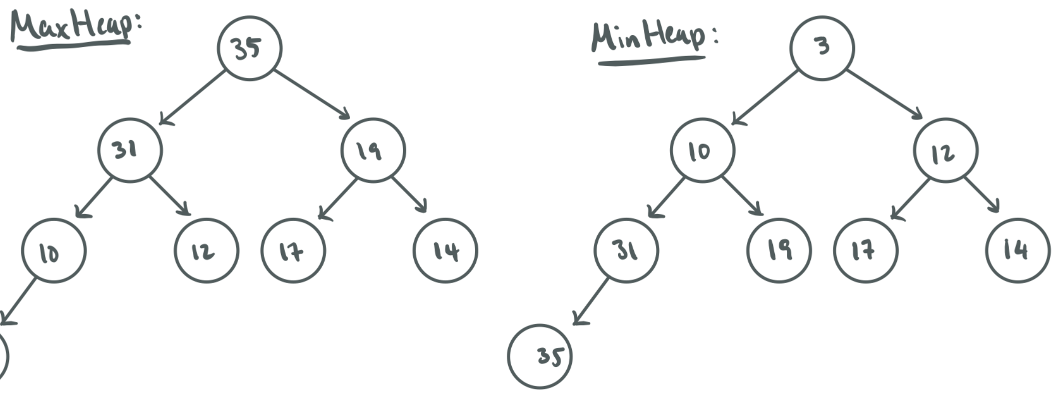


Figure 3: MaxHeap/MinHeap Diagram (3.3.2)

MAXHEAP CLASS Similar to the Queue ADT, differences in the enqueue and dequeue functions to maintain **heap** property

```
public class MaxHeap<T> {
   private T[] queue;
   private int size;

   public MaxHeap(Class<T> clazz, int maxSize) {
      queue = (T[]) Array.newInstance(clazz, maxSize);
      size = 0;
   }
   public boolean isEmpty() {
      return (size == 0);
   }
   public boolean isFull() {
      return (size == queue.length);
   }
}
// enqueue()/dequeue() functions shown later
```

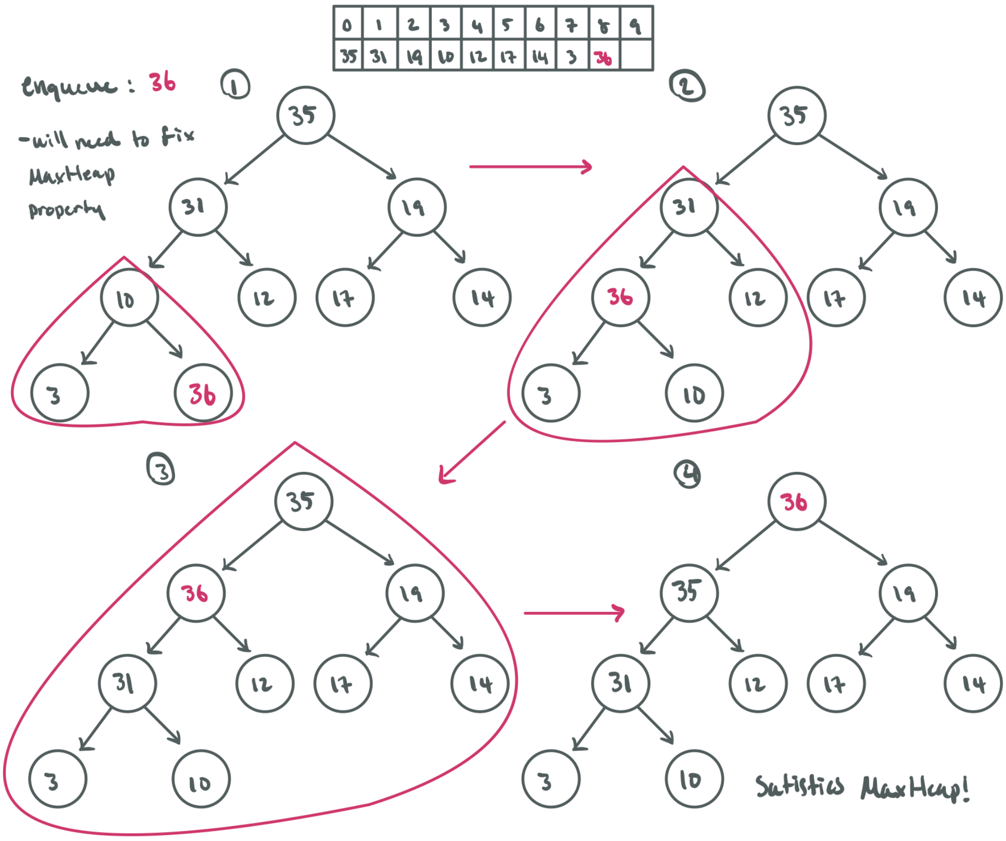
3.3.3 *Insertion*

Definition. Inserting an element (Figure 4 on the next page)

- Must keep the tree **complete**
- Must keep the **max heap** property
- **Complexity of *Enqueue***
  - *Worst-case*: added node percolates from leaf to root
  - Since the tree is **complete**, height is $O(\log(n))$
  - Hence, **enqueue** is $O(\log(n))$

Enqueue:

```
public void enqueue(T item) {
   queue[size] = item;

   // Fix heap
   int loc = size;
   int parent = (loc - 1)/2;
   while (loc > 0 && queue[loc].compareTo(queue[parent]) > 0) {
      swap(loc, parent);
      loc = parent;
      parent = (loc - 1)/2;
   }
   size++;
}
```



0	1	2	3	4	5	6	7	8	9
35	31	19	10	12	17	14	3	36	

Figure 4: MaxHeap: Enqueue

Correctness: **Loop Invariant** for Enqueue

Proof. □

3.3.4 *Deletion*

Definition. Deleting an element (Figure 5 on the following page)

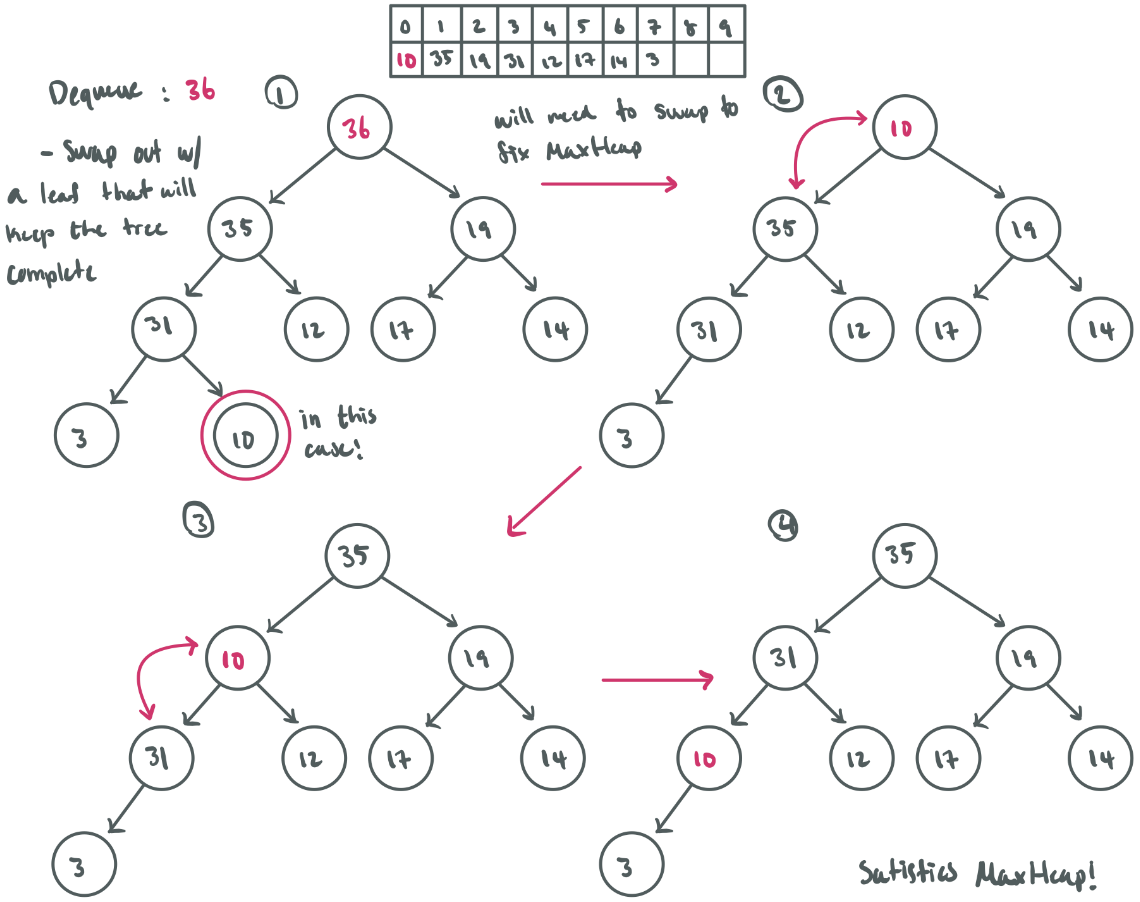
- Must keep the tree **complete**
- Must keep the **max heap** property
- This process is called *sinking*, where the leaf moves to location (index) 0
- We then *sink* the leaf back to its respective location
- **Complexity of *Dequeue***
  - Deleted root "hole" always sinks to a leaf
  - Since the tree is complete, height is $O(\log(n))$
  - Hence, **dequeue** is $O(\log(n))$

Dequeue

```
public T dequeue() {
   T max = queue[0];
   queue[0] = queue[size - 1];
   sink(0);
   queue[size - 1] = null;
   size--;
   return max;
 }
```

Sink

```
int swapWith
int left = 2*loc + 1;
int right = 2*loc + 2;
T tmp = queue[loc];

if (left > (size - 1)) { return;} // Reached a leaf
else if (left == size - 1) { // Node with one child
   if (tmp.compareTo(queue[left] < 0)) {
      swap(loc, left);
   }
} else { // Node with two children
   if (queue[left].compareTo(queue[right]) < 0) {
      swapWith = right;
   } else {
      swapWith = left;
   }
   if (tmp.compareTo(queue[left]) < 0) {
      swap(loc, swapWith);
   }
   sink(swapWith);
}
```



0	1	2	3	4	5	6	7	8	9
10	35	19	31	12	17	14	3		

Dequeue : 36
1
- Swap out w/ a leaf that will keep the tree complete
36
35
19
31
12
17
14
3
10
in this case!
will need to swap to fix MaxHeap
2
3
4
Satisfies MaxHeap!

Figure 5: MaxHeap: Dequeue

Correctness: **Bound Function** for Dequeue

Proof. Prove that sink() maintains heap property □

3.4 Hash Table

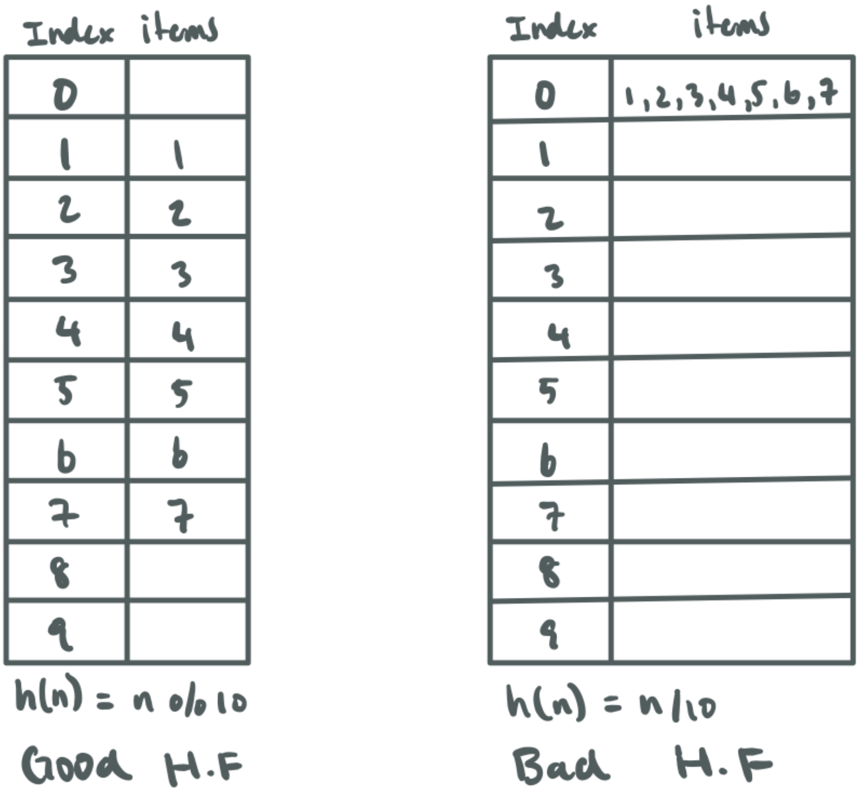
3.4.1 *Properties*

Definition. *Hash Table ADT*

- Allows adding, removing, searching for items in $O(1)$ on average
- Implementation is called *hashing*
- Objects are mapped to a hash table based on a value calculated by a **hash function** using an object's key
  - e.g. ID of a Student object as a *key*
- Hash function:

$$h(x) : U \to \{0, 1, ..., m - 1\}$$

  - $U \to$ set of all Objects
  - $|U| = n$
  - $m \to$ hash table size (buckets)
  - Typically $n > m$ (much larger)
  - Given an object $x \in U, h(x)$ maps x to a bucket in the hash table
- Performance of the hash table is dependent on the quality of the hash function
- A **Good** hash function distributes items *evenly* among buckets in the hash table



Index	items
0	
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	
9	

$h(n) = n \% 10$

Good H.F

Index	items
0	1,2,3,4,5,6,7
1	
2	
3	
4	
5	
6	
7	
8	
9	

$h(n) = n/10$

Bad H.F

Figure 6: Good vs Bad Hash Functions

Collisions

- Occurs when two objects map to the same hash value
  - Objects o_1 and $o_2 \to h(o_1) = h(o_2)$
  - Unavoidable since $n > m$
- Can be dealt with:
  - Chaining
  - Probing/Open addressing
  - Double hashing

3.4.2 *Chaining*

Definition. *Chaining*

- Group items with the same hash value (via a hash function) into a bucket (AKA a chain)
- e.g. Figure 6 on the previous page
- **Time Complexity**:
  - *Worst-case*: Hash table becomes a single LL if elements are mapped to one bucket - $O(n)$ search
  - *Best-case*: Search can become $O(1)$, but is actually $O(n/m)$
    - m size of the table, n number of elements to be hashed

Hashable Objects

```
public class HashableObject implements Hashable {
   private String key;
   // Other aspects of the object
   public HashableObject(String s) {
      key = new String(s);
   }
   public String key() {
      return key;
   }
}
```

3 Hash Functions, each used for different scenarios

- *Hash Function 1*
- Fast, but not good for large table sizes

```
public class HF1 extends HF {
   public int hash(String s, int size) {
      int hashVal = 0;
      for (int i = 0; i < s.length; i++) {
         hashVal += s.charAt(i);
         // Returns int code, e.g. A 65, B 66, etc
      }
      return hashVal % size;
   }
}
```

- *Hash Function 2*
- Fast, generally better than HF1, but horrible distributions of values where the first 3 chars are frequent

```
public class HF2 extends HF {
   public int hash(String s, int size) {
      int hashVal = 0;
      for (int i = 0; i < s.length; i++) {
         if (i == 0) { hashVal += s.charAt(0);}
         else if (i == 1) {hashVal += 27 * s.charAt(1);}
         else if (i == 2) {hashVal += 729 & s.charAt(2);}
      }
      return hashVal % size;
   }
}
```

- *Hash Function 3*
- Fast and has good distribution

```
public class HF3 extends HF {
   public int hash(String s, int size) {
      int hashVal = 0;
      for (int i = 0; i < s.length; i++) {
         hashVal += 37 * hashVal + s.charAt(i);
      }
   hashVal % size;
   if (hashVal < 0) {hashVal += size;}
   return hashVal;
   }
}
```

Hash Table implementation with Chaining

- The interface includes clear(), add(), remove(), and contain()

```
public class HashTableSC<T extends Hashable> implements HashTableInterface<T> {
   private LinkedList<T>[] hashTable;
   private HashFunction f;

   public HashTableSC(HashFunction f, int m) {
      hashTable = new LinkedList[m];
      this.f = f;
   }
   // Clear
   public void clear() {
      for (int i = 0; i < hashTable.length; i++) {
         hashTable[i] = null;
      }
   }
   // Insertion
   public void add(T item) {
      int i = f.hash(item.key(), hashTable.length);
      if (hashTable[i] == null) {
         hashTable[i] = new LinkedList<T>();
      }
      hashTable[i].add(item);
   }
   // Deletion
   public void remove(T item) {
      hashTable[f.hash(item.key(), hashTable.length)].remove(item.key());
      // Code before .remove will give you the Chain
   }
   // Search
   public boolean contains(T item) {
      int i = f.hash(item.key(), hashTable.length);
      if (hashTable[i] == null) {return false; }
      else {return hashTable[i].contains(item); }
   }
}
```

- **Correctness**: follows the correctness of regular LL operations
- **Time Complexity** (with Chaining):
  - *Load factor* λ of a hash table is the average length of chains

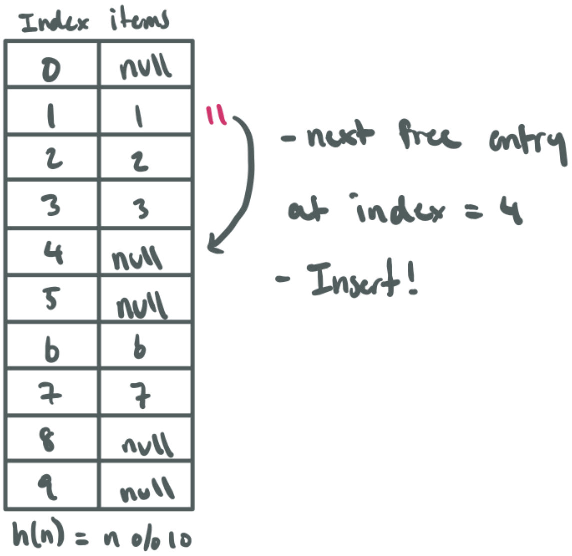
$$(n_0 + n_1 + ... + n_{m-1})/m = n/m = \lambda$$

  - Search analysis: $O(1 + \lambda)$, but can be proven $\Theta(1 + \lambda)$
  - Insertion analysis: **Average**: $\Theta(1)$, **W.C**: $O(n)$
  - Deletion analysis: **Average**: $O(1 + \lambda)$, **W.C**: $O(n)$

3.4.3 *Open Addressing*

Definition. Open Addressing

- Method for handling collisions
- All elements are stored in the hash table
- When *collision* occurs, probe the table until a free (null) entry is found



Index	items
0	null
1	1
2	2
3	3
4	null
5	null
6	6
7	7
8	null
9	null

Figure 7: Insert 11 in hash table (via hash function)

- **Linear Probing**
  - Prior to inserting 11 at index 4, we had to go through previous indicies to check if they were free
  - That is → $h(k)+0, h(k)+1, h(k)+2, h(k)+3$
  - Can be represented by: $< h(k), 0 >, < h(k), 1 >, < h(k), 2 >, ...$
  - This is the **probing sequence**, which gives us a new hashing function

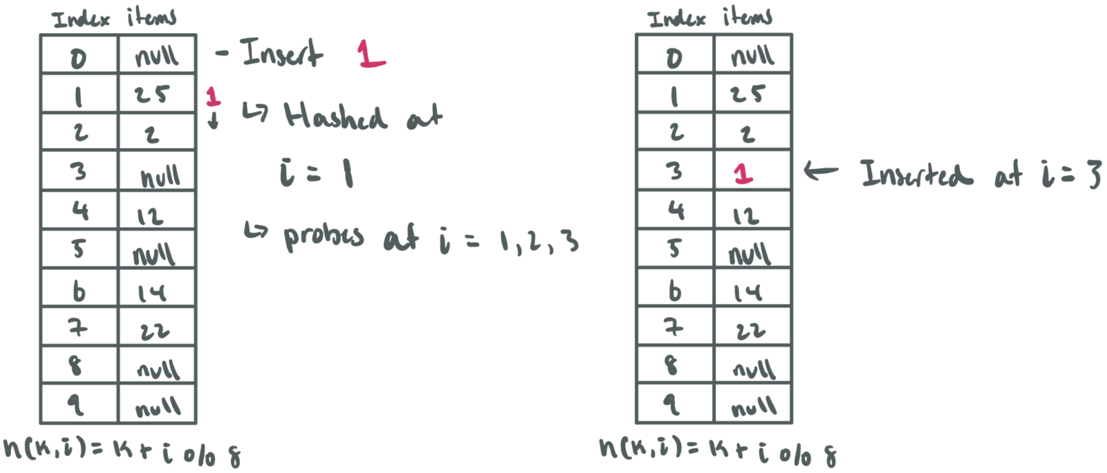
$$h'(k) = h(k) + (i \mod 10)$$

  - This is called **Linear Probing**
    - Variations: Quadratic, double-hashing, etc.

Algorithms for Linear Probing Example of Insertion

Algorithm 3: Inserting a key *k*

```
i = 0;
while $i < m$ do
    $j = h(k, i)$;
    if ($T[j] == null$ or $T[j] == Deleted$) then
        $T[j] = k$;
        return j;
    $i = i + 1$;
end
Throw a TableFullException;
```



Index	items
0	null
1	25
2	2
3	null
4	12
5	null
6	14
7	22
8	null
9	null

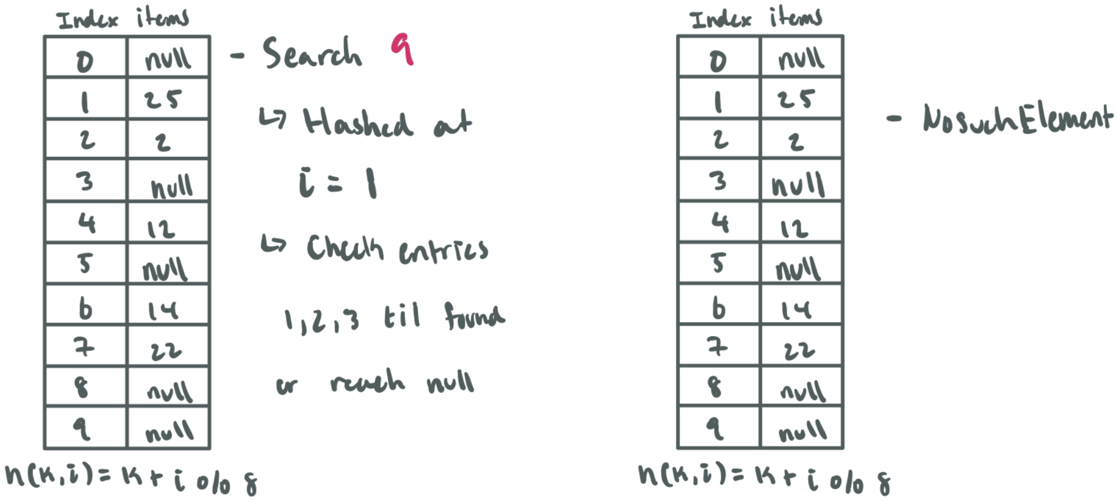
Index	items
0	null
1	25
2	2
3	1
4	12
5	null
6	14
7	22
8	null
9	null

Figure 8: Insert 1

Algorithm 4: Searching a key *k*

```
i = 0;
while i < m do
    j = h(k, i);
    if (T[j] == k) then
        return j;
    else if T[j] == null then
        Throw a NoSuchElementException;
    i = i + 1;
end
```

Example of Searching



Index	items
0	null
1	25
2	2
3	null
4	12
5	null
6	14
7	22
8	null
9	null

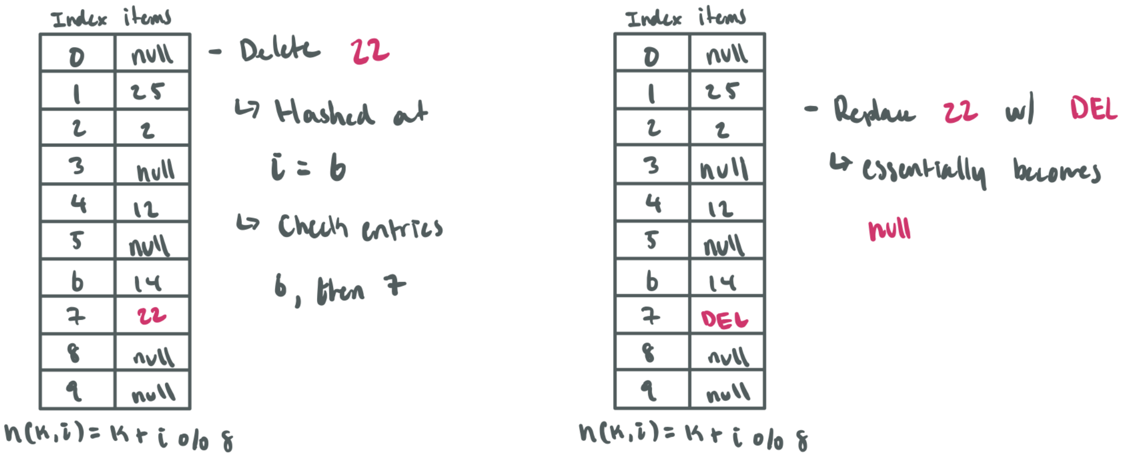
Index	items
0	null
1	25
2	2
3	null
4	12
5	null
6	14
7	22
8	null
9	null

Figure 9: Search 9

Algorithm 5: Deleting a key *k*

```
i = 0;
while i < m do
    j = h(k, i);
    if (T[j] == k) then
        T[j] == deleted;
        return;
    else if (T[j] == null) then
        Throw a NoSuchElementException;
    i = i + 1;
end
Throw a NoSuchElementException;
```

Example of Deletion



Index	items
0	null
1	25
2	2
3	null
4	12
5	null
6	14
7	22
8	null
9	null

Index	items
0	null
1	25
2	2
3	null
4	12
5	null
6	14
7	DEL
8	null
9	null

Figure 10: Delete 22

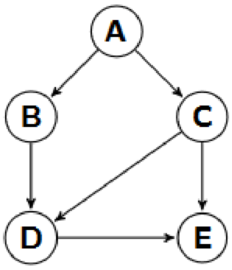
3.5 Graphs

Relations

- $S = \{1, 2, 3\}$
- $S \times S = \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)\}$
- $(S, <) = \{(1,2), (1,3), (2,3)\}$
- $(S, \geqslant) = \{(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)\}$

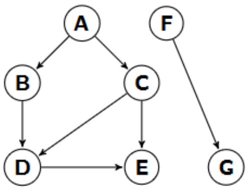
Connected

- An undirected Graph, G, is connected if there exists a path between each pair of vertices in G
- Node E does not have a directed path outward



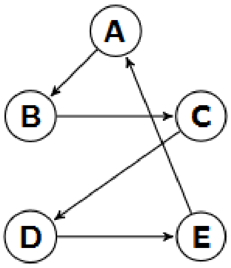
Weakly Connected

- G is weakly connected if the undirected underlying graph obtained by replacing every directed edge by an undirected edge is connected



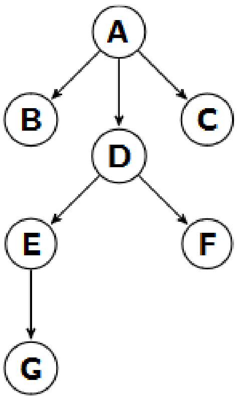
Strongly Connected

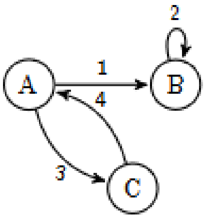
- G is strongly connected if there is a directed path between each pair of vertices in G
- There is a directed path from every node to another node in G



Directed Free Tree

- Is a connected acyclic directed graph
- A tree is a free tree with:
  - Exactly one vertex has no edges going outward (the root)
  - An ordering of the children (adjacent-from) of each vertex





Adjacency Matrix

- Represented as an $n \times n$ array
- Good for small graphs
- Can result in sparse graphs at large scales
- If the graph is undirected, then the adjacency matrix will be mirrored over $y = x$

	A	**B**	**C**
A	0	1	3
B	0	2	0
C	4	0	0

Adjacency List

- Represented as an array (or LL) where each element is a pointer to the head of a different LL
- Solves the problem for sparse graphs
- Better for larger graphs
- **Note**: AL[0] corresponds to **A**, AL[1] to **B**, AL[2] to **C**

$$AL[0] \to [(B, 1) \to (C, 3)]$$

$$AL[1] \to [(B, 2)]$$

$$AL[2] \to [(A, 4)]$$

3.6 Dijkstra's Algorithm

3.6.1 *Algorithm*

```
int[] Dijkstra(Graph g, int start) {
   Set<Integer> seen = {start};
   for (each v in g.vertices) {
      distance[v] = g.adjacencyMatrix[start][v]; // Distance array of initial cost of paths
   }                                    // from start to all other vertices
   repeat(n - 1 times) {
      Choose a vertex w in vertices not seen where distance[w] is the minimum currently;
      add w to seen;
      for (each v not seen) {
         distance[v] = min(distance[v], distance[w] + g.adjacencyMatrix[w][v]);
      }  // Going from start to v could be cheaper through w
   }
   return distance;
}
```

Consider the following:

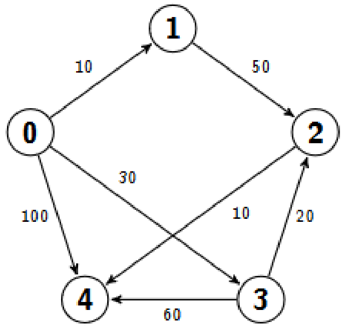


Figure 11: Example of a Directed Weighted Graph

Start of 1st Iteration

- w = 1 since distances[w] is the minimum value the is not marked as seen (Node 0 is marked as seen initially)
- $seen = \{0\}$
- $distance = \{0, 10, \infty, 30, 100\}$

	0	1	2	3	4
0	0	10	∞	30	100

- $path = \{0, 0, 0, 0, 0\}$

0	0	0	0	0

- Values at each index of path represent the current node we are at
- Each index represents the next node in the shortest path that we can get to from the current node

End of 1st Iteration

- w = 1 (still)
- $seen = \{0, 1\}$

- $distance = \{0, 10, 60, 30, 100\}$

	0	**1**	**2**	**3**	**4**
0	0	10	∞	30	100
1	0	10	60	30	100

- $path = \{0, 0, 1, 0, 0\}$
- Now that we have seen node 1, we know it is possible to get to node 2 by traversing to 1 and then 2 which has a weight of 10 + 50 = 60
- Since node 1 only points to 2, the rest of the distances do not change

0	0	1	0	0

- It is possible to get to all nodes through 0 except node 2 which is from 0 → 1 → 2 currently
- Currently, the shortest path to node 2 is through node 1

Start of 2nd Iteration

- w = 3 since distances[w] is the minimum value of the nodes not seen after the first iteration (30 is the smallest of $\{60, 30, 100\}$)
- $seen = \{0, 1\}$ (still)
- $distances = \{0, 10, 60, 30, 100\}$ (still)
- $path = \{0, 0, 1, 0, 0\}$ (still)

End of 2nd Iteration

- w = 3 (still)
- $seen = \{0, 1, 3\}$
- $distances = \{0, 10, 50, 30, 90\}$

	0	**1**	**2**	**3**	**4**
0	0	10	∞	30	100
1	0	10	60	30	100
2	0	10	50	30	90

- Shortest path to node 2 is now through node 3 which has a cost of 30 + 20 = 50
- Shortest path to node 3 is unchanged
- Shortest path to node 4 is updated since 90 < 100 and this is cost is calculated by 30 + 60
- $path = \{0, 0, 3, 0, 3\}$

0	0	3	0	3

- Shortest path to node 2 is changed from 0 → 1 → 2 to 0 → 3 → 2
- Shortest path to node 4 is changed from 0 → 4 to 0 → 3 → 4

Start of 3rd Iteration

- w = 2 since distances[w] is the minimum cost of the nodes we have not seen. That is, distances[2] = 50 is the minimum of $\{50, 90\}$
- $seen = \{0, 1, 3\}$ (still)
- $distances = \{0, 10, 50, 30, 90\}$ (still)
- $path = \{0, 0, 3, 0, 3\}$ (still)

End of 3rd Iteration

- w = 2 (still)
- $seen = \{0, 1, 3, 2\}$
- $distances = \{0, 10, 50, 30, 60\}$

	0	**1**	**2**	**3**	**4**
0	0	10	∞	30	100
1	0	10	60	30	100
2	0	10	50	30	90
3	0	10	50	30	60

- The only node we can to from node 2 is 4 so all weights except 4 are unchanged
- The cost to node 4 is updated as the cost to node 2 plus the cost to get from 2 → 4 which is 50 + 10 = 60 and since 60 < 90, it is updated
- $path = \{0, 0, 3, 0, 2\}$

0	0	3	0	2

- The paths array is also updated for shortest path to node 4 is changed to 0 → 3 → 2 → 4

Start of 4th Iteration

- w = 4 since it is the only node that has not yet been seen so it is certainly a minimum of what is still unseen
- $seen = \{0, 1, 3, 2\}$ (still)
- $distances = \{0, 10, 50, 30, 60\}$ (still)
- $path = \{0, 0, 3, 0, 2\}$ (still)
- However, node 4 does not have any edges directed outwards thus there is nothing more that can be done

End of 4th Iteration

- $seen = \{0, 1, 3, 2, 4\}$
- Everything else the same

Generating the Shortest Path

- Let our starting point be some node, start
- Let our destination point be some node, dest, then set $current = dest$
- Now, while current $\neq$ start then current = paths[current] to get the next node in the shortest path working our way backwards from the destination to the start
- **Ex**:
  - Let start = 0, dest = 4
  - Lets record each node we see in a dynamic array, $arr = \{\}$
  - **Iteration 1**
    - current = 4
    - $current \neq 0$
    - Then, current = paths[4] = 2, $arr = \{4\}$
  - **Iteration 2**
    - current = 2

    - $\mathrm{current} \neq 0$
    - Then, current = paths[2] = 3, $\mathrm{arr} = \{2, 4\}$
  - **Iteration 3**
    - current = 3
    - $\mathrm{current} \neq 0$
    - Then, current = paths[3] = 0, $\mathrm{arr} = \{3, 2, 4\}$
  - **Iteration 4**
    - current = 0
    - $\mathrm{current} = 0$ is true
    - Add it: $\mathrm{arr} = \{0, 3, 2, 4\}$
  - Now, the shortest path from node 0 to node 4 has been obtained

3.6.2 *Correctness*

Definition. *Loop Invariant*

- ***Initialization***
  - Before the first iteration, S_0 only contains start and all special paths that exist from start to other vertices that have a single edge
  - All existing such special paths are the shortest and $D_0[v]$ contains their cost
  - Otherwise, $D_0[v] = \infty$
- ***Maintenance***
  - Assume $D_{i-1}[v]$ contains the cost of the shortest special path from start to v
  - We show that $D_i[v]$ contains the cost of the shortest special path from start to v
  - In iteration i, we add w to Si where $D_i[w]$ is the minimum
  - For a vertex v, $D_i[v]$ will be updated if it is shorter to go from start to v through w
- ***Termination***
  - The repeat loop will iterate n - 1 times
  - Hence, $S_{n-1} = G.$ vertices (All vertices have been seen)
  - After this last iteration, $D_{n-1}[v]$ is the cost of the shortest special path from start to v if such a path exists
  - If there is not path from start to v, then $D[v] = \infty$

4 SORTING ALGORITHMS

4.1 Bubble Sort

4.1.1 *Algorithm*

```
// Bubble Sort in Sorting
public void sort() {
   for (i = 0; i < size; i++) {
      for (j = size - 1; j > i; j--) {
         if (list[j].compareTo(list[j-1]) < 0) {
            T tmp = list[j];
            list[j] = list[j-1];
            list[j-1] = tmp;
         }
      }
   }
}
// Bubble Sort in BoundedList
public static void sort(BoundedList list) {
   for (i = 0; i < list.size(); i++) {
      for (j = list.size() - 1; j > i; j--) {
         if (list.getItem(j).compareTo(list.getItem(j-1)) < 0) {
            list.swap(j, j-1);
         }
      }
   }
}
// Main algorithm
Bubble-Sort(L = list of items, n = size) {
   for (i = 0; i < n; i++) {
      for (j = n - 1; j > i; j--) {
         if (L[j].key < L[j-1].key) {
            swap(L[j], L[j-1]);
         }
      }
   }
}
// Where swap() is...
swap(item n1, item n2) {
   tmp = n1;
   n1 = n2;
   n2 = tmp;
}
```

Example:

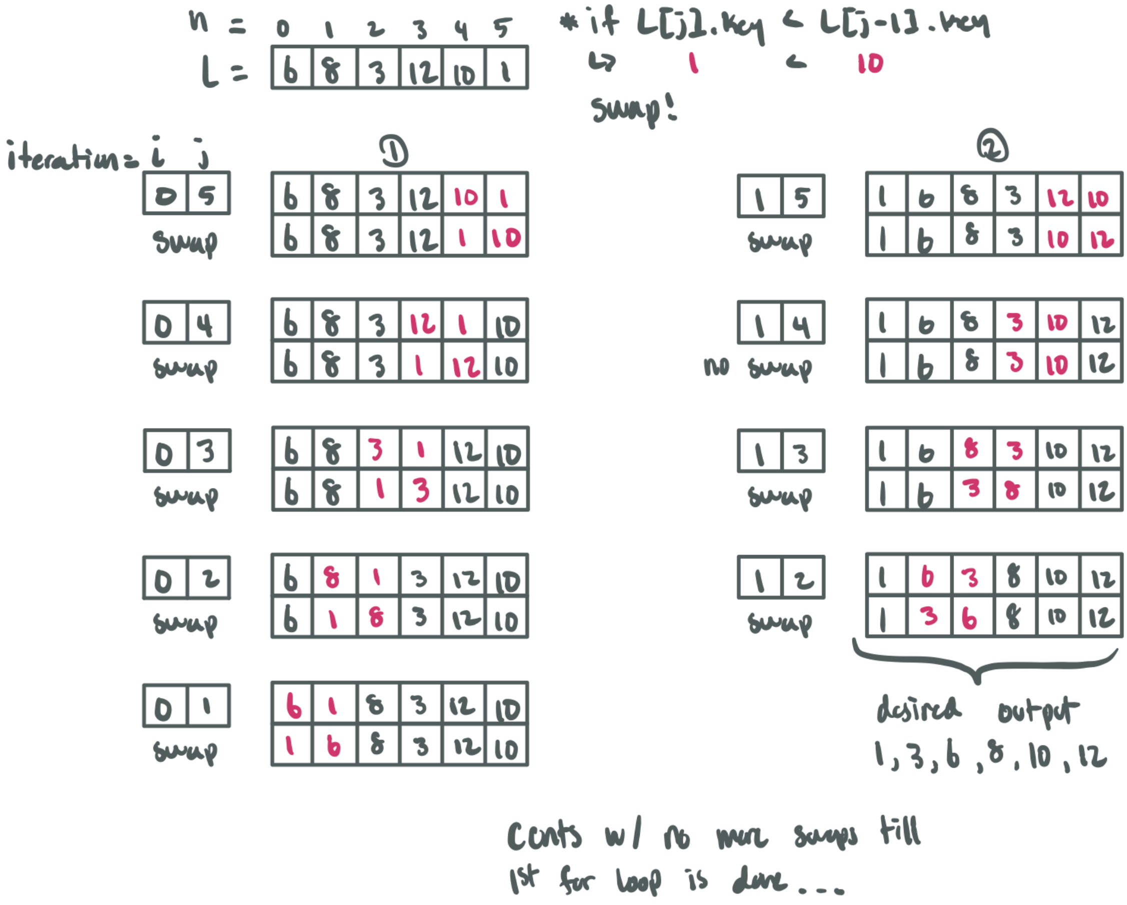


Figure 12: Bubble sort following the main algorithm

4.1.2 *Time Complexity*

Definition. *Time Complexity*

- **swap()**: $\Theta(1)$
- **j-loop**: executed $n - i - 1$ times
- **i-loop**: executed n times
- **Bubble-Sort**: $\Sigma_{i=0}^{n-1} n - i - 1$

$$\Sigma_{i=0}^{n-1} n - i - 1 = (n-1) + (n-2) + ... + 1 + 0 = \frac{n^2 - n}{2}$$

- **Hence**, Bubble-Sort is quadratic, $\Theta(n^2)$

4.1.3 *Correctness*

Definition. *Loop Invariant*

- **Inner Loop**
  - *Initialization*:
    - Initially, $j = n - 1$ and $L[n - 1...n - 1]$ is a singleton
      - LI is vacuously satsified
  - *Maintenance*:
    - Assume LI is correct before iteration $j = k$
    - That is, $L[k]$ is the min in $L[k...n - 1]$
    - In iteration $j = k$
      - If $L[k].key < L[k - 1].key$, then swap$(L[k], L[k - 1])$, making $L[k - 1]$ the new min in $L[k - 1...n - 1]$
      - Else, item being considered $L[k - 1].key \leqslant L[k].key$, meaning that $L[k - 1]$ is the min in $L[k - 1...n - 1]$
      - Hence, before the next iteration $j = k - 1$, $L[k - 1]$ is the min in $L[k - 1...n - 1]$
  - *Termination*:
    - Last iteration of the loop is when $j = i - 1$
    - Hence, by LI, $L[i]$ is the min item in $L[i...n - 1]$
- **Outer Loop**
  - *Initialization*:
    - Before the first iteration, $i = 0$ and $L[0...i - 1]$ is empty and LI is vacuously true
  - *Maintenance*:
    - Assume LI is correct before iteration $i = k$
    - Hence, $L[0...k - 1]$ is a sorted list of the smallest *k* items in L
    - In iteration $i = k$, by the LI of the **inner loop**:
      - $L[k]$ is the min item in $L[k...n - 1]$
    - Since $L[0...k - 1]$ has the smallest *k* items of L in sorted order,
    - $L[0...k]$ will have the smallest $k + 1$ items in L in sorted order
  - *Termination*:
    - Last iteration of the loop is when $i = n - 1$
    - Hence, when the loop terminates, $L[0...n - 2]$ is a sorted permutation of the smallest $n - 1$ elements in L
    - Since $L[n - 1]$ was considered the last iteration and compared with $L[n - 2]$, it must be the case that $L[n - 1]$ is the largest item in L
    - Therefore, when the loop terminates, $L[0...n - 1]$ is a sorted permutation of the original elements in L

4.2 Selection Sort

4.2.1 *Algorithm*

```
// Selection sort in Sorting
public static void selectionSort(BoundedList list) {
   for (int i = 0; i < list.size(); i++) {
      int minIndex = list.getMinIndex(i, list.size()-1);
      list.swap(i, minIndex);
   }
}
// Main algorithm
Selection-Sort(L = list of items, n = size) {
   for (i = 0; i < n; i++) {
      min = Find-Min(L[i...n-1]);
      swap(L[i], min);
   }
}
// Find minimum
Find-Min(L = list of items, n = size) { // Must be non-empty
   currMin = L[0];
   for (i = 1; i < n; i++) {
      if (L[i] < currMin) {
         currMin = L[i];
      }
   }
   return currMin;
}
```

Example:

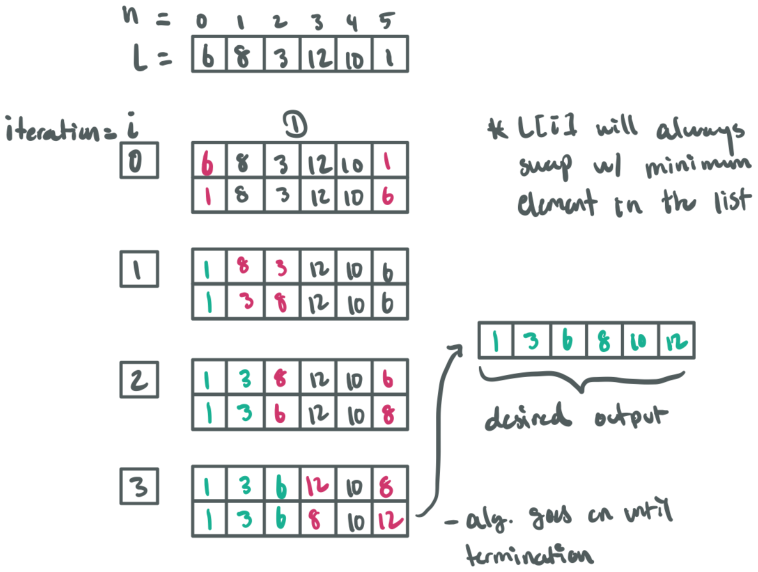


Figure 13: Selection sort following main algorithm

4.2.2 *Time Complexity*

Definition. *Time Complexity*

- **loop**: executed $n - 1$ times

- **Find-Min()**: $\Theta(n)$ on all cases
  - List is reduced by 1 after every iteration
  - 1st: $|L| = n$
  - 2nd: $|L| = n - 1$
  - ith: $|L| = n - i - 1$
- **Selection-Sort**: $\Sigma_{i=1}^{n-1} n - i - 1$

$$\Sigma_{i=1}^{n-1} n - i - 1 = (n-2) + ... + 1 = \frac{n^2 - 2n + 2}{2}$$

- **Hence**, Selection-Sort is quadratic, $\Theta(n^2)$

4.2.3 *Correctness*

Definition. *Loop Invariant*

- **Find-Min**
  - ***Initialization***
    - Before the first iteration, *currMin* is $L[0]$, which is the minimum item in the singleton $L[0...0]$
  - ***Maintenance***
    - Assume LI is correct before iteration $i = k$
    - Hence, *currMin* is the min item in $L[0...k-1]$
    - In iteration $i = k$
      - If $L[k] < currMin$, then $L[k]$ is the min in $L[0...k]$ and *currMin* is updated accordingly
      - Else, $L[k] \geqslant currMin$, *currMin* is still the min item in $L[0...k]$
    - Hence, LI is also correct after iteration $i = k$
  - ***Termination***
    - Last iteration of the for loop is $i = n - 1$
    - Thus, when loop terminates, *currMin* is the min item in $L[0...n-1]$
- **Selection-Sort**
  - ***Initialization***
    - Before the first iteration, the singleton $L[0... - 1]$ is empty, LI being vacuously true
  - ***Maintenance***
    - Assume LI to be correct before iteration $i = k$
    - Hence, $L[0...k-1]$ is a sorting of $S[0...k-1]$
      - In iteration $i = k$, min of $L[0...k-1]$ is swapped with $L[k]$
      - Since $L[k]$ was not picked as the min in a previous iteration, it must be the case that each item in $L[0...k-1]$ is not bigger than $L[k]$
    - Hence, $L[0...k]$ is sorted
    - Since the only change introduced to $L[0...k]$ is swapping existing elements of $S[0...k]$, $L[0...k]$ is a sorting of $S[0...k]$
    - Therefore, LI holds for the end of iteration $i = k$
  - ***Termination***
    - Last iteration of the for loop is $i = n - 1$
    - Min of $L[n-2...n-1]$ is moved up to $L[n-2]$ and now $L[n-1]$ has the largest item in the list
    - Hence, when the loop terminates, $L[0...n-1]$ will be sorted

4.3 Insertion Sort

4.3.1 *Algorithm*

```
// Insertion Sort in Sorting
public static void sort(BoundedList list) {
   for (i = 1; i < list.size(); i++) {
      for (j = i; j > 0; j--) {
         if (list.getItem(j).compareTo(list.getItem(j-1)) < 0) {
            list.swap(j, j-1);
         }
      }
   }
}
// Main algorithm
Insertion-Sort(L = list of items, n = size) {
   for (i = 1; i < n; i++) {
      for (j = i; j > 0; j--) {
         if (L[j].key < L[j-1].key) {
            swap(L[j], L[j-1]);
         }
      }
   }
}
// Where swap() is...
swap(item n1, item n2) {
   tmp = n1;
   n1 = n2;
   n2 = tmp;
}
```

Example:

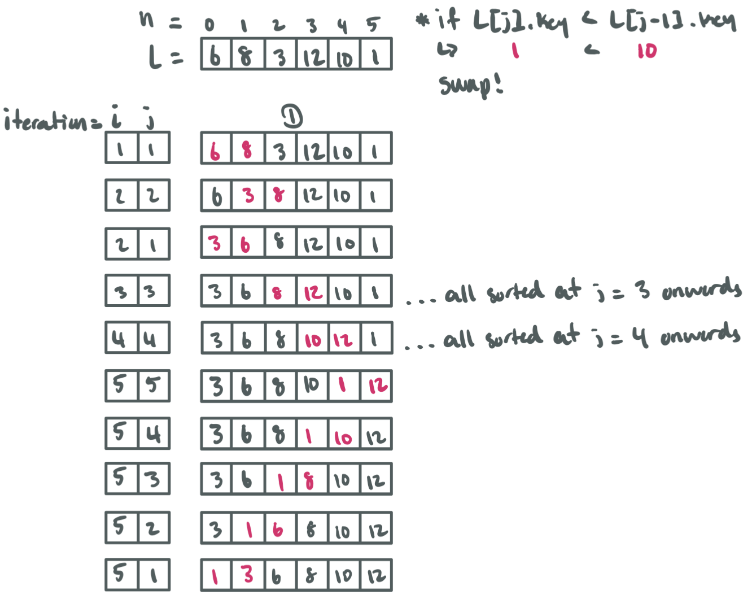


Figure 14: Insertion sort following the main algorithm

4.3.2 *Time Complexity*

Definition. *Time Complexity*

- **swap()**: $\Theta(1)$
- **j-loop**: executed i times
- **i-loop**: executed $n-1$ times, $\Theta(n)$
- **Insertion-Sort**: $\Sigma_{i=1}^{n-1} n-i-1$

$$\Sigma_{i=1}^{n-1} n-i-1 = (n-1)+(n-2)+...+1+0 = \frac{n^2-n}{2}$$

- **Hence**, Insertion-Sort is quadratic, $\Theta(n^2)$

4.3.3 *Correctness*

Definition. *Loop Invariant*

- **Inner Loop**
  - ***Initialization***:
    - * Initially, $j = i = 1$, $L[0...0]$ is a sorted permutation of $S[0]$
      - · LI is vacuously satsified
  - ***Maintenance***:
    - * Assume LI is correct before iteration $j = k$
    - * Hence, $L[k-1...i-1]$ is a sorted permutation of $S[k-1...i-1]$
    - * We show LI still holds at the end of this iteration before $j = k-1$
      - · Loop compares $L[k-1], L[k-2]$
      - · If $L[k-2].key < L[k-1].key$, $swap(L[k-2].key, L[k-1].key)$
      - · Now, $L[k-2...i-1]$ is also a sorted permutation of $S[k-2...i-1]$
      - · Else, $L[k-2].key$ is larger than $L[k-1].key$, loop does nothing and $L[k-2...i-1]$ is still a sorted permutation of $S[k-2...i-1]$
    - * In both cases, the loop makes sure $L[k-2...i-1]$ has the same items as $S[k-2...i-1]$
    - * Hence, LI still holds at the end of the iteration
  - ***Termination***:
    - * Last iteration of the loop is when $j = 1$
    - * Always terminates because j will eventually reach 0, which is the condition for the for-loop to be false
    - * Hence, at the end of it, $L[0...i-1]$ is a sorted permutation of $S[0...i-1]$ where $i \in [1, n]$
- **Outer Loop**
  - ***Initialization***:
    - * Before the first iteration, $i = 1$ and $L[1...n-1]$ is empty and LI is vacuously true
  - ***Maintenance***:
    - * Assume LI is correct before iteration $i = k$
    - * Hence, by the inner LI, $L[0...n-2]$ is a sorted permutation of $S[0...n-2]$ after the last iteration of the outer loop
    - * Suffices to note that when $i = n-1$ (last iteration of the outer loop)
    - * If $L[n-2].key < L[n-1].key$, swap

    - Else, don't do anything
    - Hence, $L[0 ... n-1]$ is a sorting of $S[0 ... n-1]$
  - ***Termination***:
    - Last iteration of the loop is when $i = n - 1$
    - Starts with $i = 1$
    - i doesn't change anywhere other than incrementation
    - i will eventually become n
    - Hence, Insert-Sort() always terminates

4.4 Heap Sort

4.4.1 *Algorithm*

```
// LinkedList implementation
Heap-Sort(L = list, n = size) {
   max-heap H = empty;
   min-heap H = empty;
   for (i = 0; i < n; i++) {
      H.enqueue(L.remove(L.getNext()));
   }
   for (int = 0; i < n; i++) {
      L[i].append(H.dequeue());
   }
}
// Main algorithm
Heap-Sort(L = list, n = size) {
   max-heap H = empty; // MAX for descending order
   min-heap H = empty; // MIN for ascending order
   for (i = 0; i < n; i++) {
      H.enqueue(L[i]);
   }
   for (int = 0; i < n; i++) {
      L[i] = H.dequeue();
   }
}
// Heap-Sort with Heapify
Heap-Sort(int[] arr) {
   int size = arr.length - 1;

   for (int i = size; i > 0; i++) {
      swap(arr[0], arr[i]);
   }
}
Heapify(int[] arr, int size, int i) {
   int largest = i;
   int left = 2*i + 1;
   int right = 2*i + 2;

   if (left < size && arr[left] > arr[largest]) {
      largest = left;
   }
   if (right < size && arr[right] > arr[largest]) {
      largest = right;
   }
   if (largest != i) {
      swap(arr[i], arr[largest]);
   }
}
```

4.4.2 *Time Complexity*

- **Main algorithm**
  - Both loops are $O(n)$
  - Enqueue and Dequeue operations are $O(\log(n))$
  - Hence, Heap-Sort is $O(n \log(n))$

- **Heap-Sort with Heapify**
  - **W.C**: $O(n\log(n))$
  - **A.C**: $\Theta(n\log(n))$

4.4.3 *Correctness*

Definition. *Loop Invariant*

- **1st loop**
  - ***Initialization***
    - Before first iteration, $i = 0$, $H[0...i-1]$ is empty thus vacuously a min-heap of any set items
  - ***Maintenance***
    - Assume LI correct before iteration *k*, for $k \in [0, k]$
    - That is, $H[0...k-1]$ is a min-heap of the *k* items from $L[0...k-1]$
    - We want to prove that before iteration $k+1$, $H[0...k]$ is a min-heap of items in $L[0...i]$
    - The body of the loop simply enqueues $L[k]$
      - By the correctness of enqueue(), the resulting $H[0...k]$ is a min-heap
    - Furthermore, $H[0...k-1]$ contains *k* items from $L[0...k-1]$ - this step adds another item from L, $L[k]$
    - Hence, LI holds true before iteration $k+1$
  - ***Termination***
    - Loop terminates when $i = n-1$
    - Before iteration $n-1$, we know from *maintenance* that $H[0...n-2]$ is a min-heap containing all items from $L[0...n-1]$
    - Last iteration $n-1$ enqueues $L[n-1]$ into the heap
    - Therefore, $H[0...n-1]$ is a min-heap containing all items from the original list L
- **2nd loop**
  - Proof is very similar to the **1st loop**

5 ADVANCED SORTING ALGORITHMS

5.1 Merge Sort

- Idea is to merge two sorted lists
- We recursively divide it into two equal halves (or lengths differ be 1) until the collection of lists is size 1
- From there on, we merge the divided lists, continuously sorting as we do so
- Requires extra storage $O(n\log(n))$

5.1.1 *Algorithm*

```
// Main algorithm
Merge-Sort(list L, int low, int high) {
   if (low < high) {
      int mid = (low + high)/2
      Merge-Sort(L, low, mid);
      Merge-Sort(L, mid + 1, high);
      Merge(L, low, high, mid);
   }
}
// Merge
Merge(List A, int low, int high, int mid) {
   n1 = mid - low + 1;
   n2 = high - mid;

   L = list of size n1;
   for (i = 0; i < n1; i++) {
      L[i] = A[low + i];
   }
   R = list of size n2;
   for (i = 0; i < n2; i++) {
      R[i] = A[mid + 1 + i];
   }
   i = 0;
   j = 0;
   k = low;
}
// Merging two sorted lists
while (i < n1 && j < n2) {
   if (L[i].key <= R[j].key) {
      A[k] = L[i];
      i++;
   } else {
      A[k] = R[j];
      j++;
   }
   k++;
}
```

5.1.2 *Time Complexity*

5.1.3 *Correctness*

5.2 Quick Sort

1. **Divide**: Divide into two sublists based on value with index *pivot*
2. **Conquer**: Repeat dividing until sorted
3. **Combine**: Not needed (not placing items into actual sublists)

- **Partition** value will eventually end up in its appropriate position, thus QuickSort() only worries about sorting $L[start...p-1], L[p+1...end]$
- Quick sort is known to be the **fastest** sorting algorithm, but its W.C is worse than Heap and Merge sort

5.2.1 *Algorithm*

```
// Partition
int partition(list L, int start, int end) {
   p = L[end]; // Partition will be last item in array for our case
   i = start - 1;
   for (int j = 0; j < end; j++) {
      if (L[j].key <= p.key) {
         i++;
         swap(L[i], L[j]); // No need to swap if i = j
      }
   }
   swap(L[end, L[i+1]]);
   return i + 1;
}
// Main algorithm
Quick-Sort(list L, int start, int end) {
   if (start < end) {
      p = partition(L, start, end);
      Quick-Sort(L, start, p - 1);
      Quick-Sort(L, p + 1, end);
   }
}
```

Example:

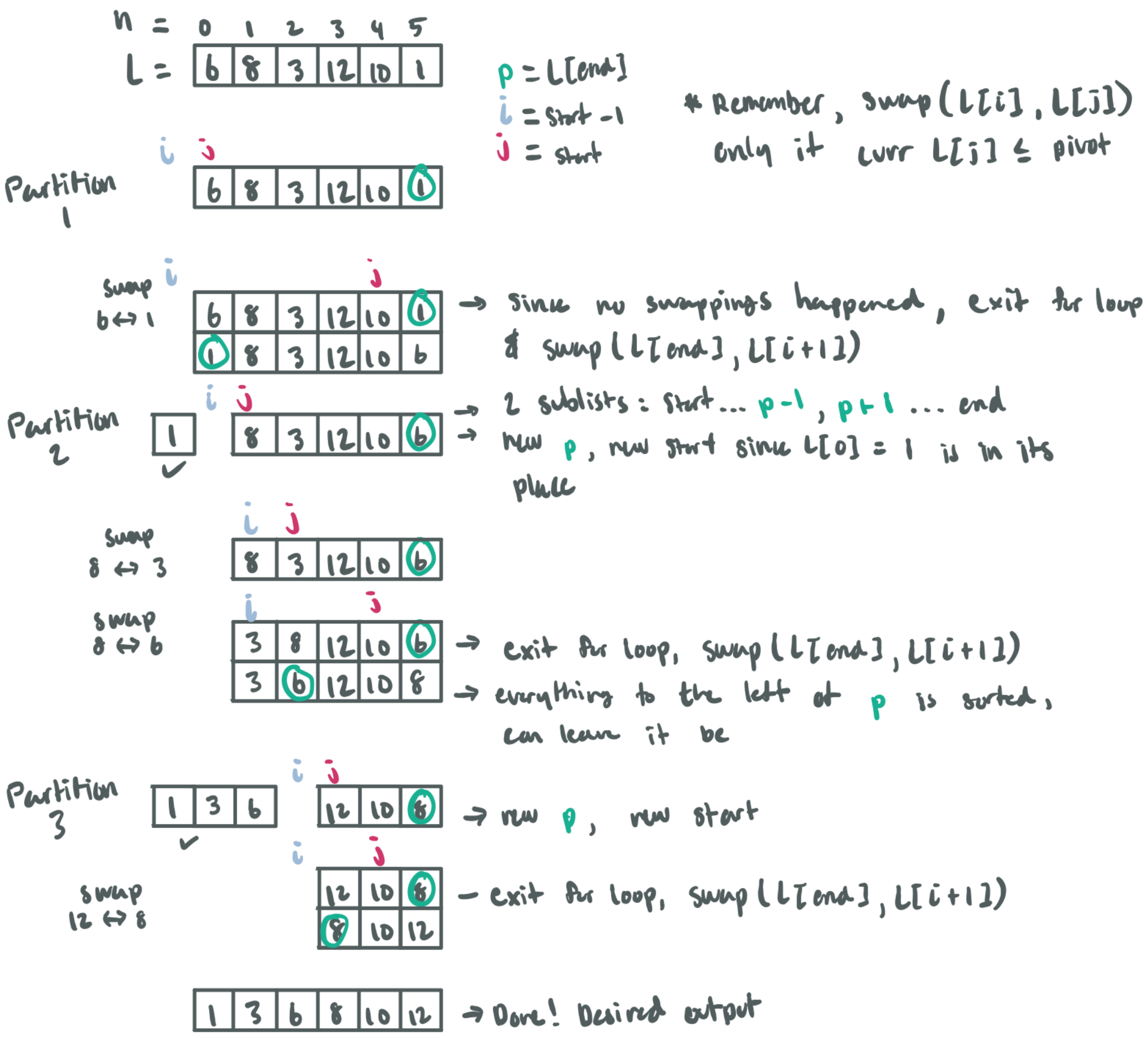

n = 0 1 2 3 4 5
L = 6 8 3 12 10 1
p = L[end]
i = start -1
j = start
* Remember, swap(L[i], L[j]) only if curr L[j] ≤ pivot
Partition 1
i j
swap 6↔1
→ Since no swappings happened, exit for loop & swap(L[end], L[i+1])
Partition 2
→ 2 sublists: start... p-1, p+1 ... end
→ new p, new start since L[0] = 1 is in its place
swap 8↔3
swap 8↔6
→ exit for loop, swap(L[end], L[i+1])
→ everything to the left of p is sorted, can leave it be
Partition 3
→ new p, new start
swap 12↔8
– exit for loop, swap(L[end], L[i+1])
1 3 6 8 10 12
→ Done! Desired output

Figure 15: QuickSort following main algorithm

5.2.2 *Time Complexity*

Definition. *Time Complexity*

- **B.C**:
  - **partition()**: $\Theta(n)$, B.C when partition creates two equal sublists (or when their lengths differ by 1)
  - **sublists**: sizes of $n/2$ and $n/2$
  - **Quick-Sort**: $\Theta(n\log(n))$, similar to binary search
- **W.C**:
  - **partition()**: $\Theta(n)$, when list is already sorted (either ascending or descending)
  - **sublists**: sizes of $n-1$ and 0 - since one of the lists is empty, recursive call is $O(1)$, other is repeated $n-1$ times
  - **Quick-Sort**: $\Theta(n^2)$

5.2.3 *Correctness*

Definition. *Loop Invariant and Mathematical Induction*

- **Partition**
  - ***Initialization***
    - * Before the 1st iteration, $i_0 = \mathtt{start} - 1, j_0 = \mathtt{start}$
      1. $L[\mathtt{start}...i_0]$ is empty
      2. $L[i_0 + 1...j_0]$ is a singleton
    - * Hence, LI is vacuously true
  - ***Maintenance***
    - * Assume LI is correct before iteration *k*. That is,
      1. $\forall y \in L[\mathtt{start}...i_{k-1}], y.key \leqslant p.key$ and
      2. $\forall z \in L[i_{k-1} + 1...j_{k-1}], z.key > p.key$
    - * In this iteration, if $L[j_k].key \leqslant p.key, i_k = i_{k-1} + 1$, swap $L[j_k], L[i_k]$
      - · That is, $\forall y \in L[\mathtt{start}...i_k], y.key \leqslant p.key$ and
      - · $\forall z \in L[i_k + 1...j_k], z.key > p.key$
    - * On the other hand, if $L[j_k].key > p.key, i_k = i_{k-1}$
      - · That is, $\forall y \in L[\mathtt{start}...i_k], y.key \leqslant p.key$ and
      - · $\forall z \in L[i_k + 1...j_k], z.key > p.key$
    - * Both hold, hence LI holds for every iteration
  - ***Termination***
    - * Last iteration is when $j = \mathtt{end} - 1$
    - * That is, for some *i*, $(\mathtt{start} - 1) \geqslant i \geqslant \mathtt{end} - 1$
      1. $\forall y \in L[\mathtt{start}...i], y.key \leqslant p.key$, and
      2. $\forall z \in L[i + 1...\mathtt{end} - 1], z.key > p.key$
    - * Last step of the algorithm is to swap $L[\mathtt{end}], L[i + 1]$, partitioning L into $L[\mathtt{start}...i + 1]$ and $L[i + 2...\mathtt{end}]$, where *pivot* is at index $i + 1$
    - * Algorithm returns *pivot*, hence it always terminates

- **QuickSort**
  - ***Claim***: Quick-Sort sorts any list $L[i...j]$ of size $j - i + 1 = n \geqslant 0$

    Proof. By mathematical induction

    - * ***Inductive Hypothesis***: Assume Quick-Sort(L, i, j) sorts $L[i...j]$ of size $j - i + 1 = k$ and $0 \leqslant k < n$
    - * ***Inductive Claim***: Quick-Sort(L, l, m) sorts $L[l...m]$ of size $m - l + 1 = k + 1$. Let *p* be the pivot for $L[l...m]$
      - · Both L[left] and L[right] are sorted by IH
      - · Partition correctness guarantees that all values in L[left] are $\leqslant p$ and values in L[right] $> p$
      - · Hence, Quick-Sort sorts any list of size $n \geqslant 0$

□

6 SEARCHING

6.1 Linear Search

6.1.1 *Algorithm*

```
// Unsorted List
public int contains(T item) {
   int i = 0;
   while (i < this.size) {
      if (item.compareTo(this.list[i]) == 0) {
         return i;
      }
      i++;
   }
   return -1;
}
// Sorted List
public int contains(T item) {
   int i = 0;
   while (i < this.size) {
      if (item.compareTo(this.list[i]) == 0) {
         return i;
      } else if (item.compareTo(list[i]) > 0) {
         return -1;
      } else {
         i++;
      }
   }
   return -1;
}
```

6.1.2 *Time Complexity*

Definition. *Time Complexity*

- **Unsorted List**
  - **B.C**: $\Theta(1)$, x is the first element in the list
  - **W.C**: $\Theta(n)$, x is not in the list or while loop iterates exactly n times
- **Sorted List**
  - **B.C**: $\Theta(1)$, x is the first element in the list
  - **W.C**: $O(n)$, x is not in the list or while loop iterates exactly n times

6.1.3 *Correctness*

Definition. *Loop Invariant*

- ***Initialization***
  - Before the first iteration, $k = 0$, $[0, 0]$ is empty, and $\text{list}[i].\text{key} \neq \text{item}.\text{key}$
  - LI holds true
- ***Maintenance***
  - If the item is found in iteration $k-1$, algorithm terminates, returning $k-1$ as required
  - So, assume item was not found in the previous iteration and there is a k^{th} iteration

- Assume LI is true before the k^{th} iteration
- **Case 1**: Since there is a k^{th} iteration, $\forall i \in [0, k] : \mathtt{list}[i].\mathtt{key} \neq \mathtt{item}.\mathtt{key}$
- **Case 2**: When $\mathtt{list}[k].\mathtt{key} \neq \mathtt{item}.\mathtt{key}$
- **1** and **2** $\rightarrow \forall i \in [0, k+1] : \mathtt{list}[i].\mathtt{key} \neq \mathtt{item}.\mathtt{key}$
- Otherwise, if in the k^{th} iteration, $\mathtt{list}[k].\mathtt{key} = \mathtt{item}.\mathtt{key}$, there will be no $k^{th} + 1$ iteration since algorithm terminates, returning *k* as required
- Hence, LI holds true at the end of the k^{th} iteration

- ***Termination***
  - While loop terminates if
    - **1.** $\mathtt{list}[i].\mathtt{key} = \mathtt{item}.\mathtt{key}$, where *i* is returned as required
    - **2.** $i = \mathtt{list}.\mathtt{size}$, -1 is returned as required
  - Hence, LI holds true when algorithm terminates

6.2 Binary Search

Only works for sorted lists, typically with an array-implementation

6.2.1 *Algorithm*

```
// Recursive
Binary-Search(item x, item[] L, int low, int high) {
   if (low > high) {
      return - 1;
   }
   int mid = (low + high)/2;
   if (x.key == L[mid].key) {
      return mid;
   } else if (x.key < L[mid].key) {
      return Binary-Search(x, L, low, mid - 1);
   } else {
      return Binary-Search(x, L, mid + 1; high);
   }
}
// Iterative
Binary-Search(item x, item[] L) {
   int low = 0;
   int high = L.size - 1;

   while (low <= high) {
      int mid = (low + high)/2;
      if (x.key == L[mid].key) {
         return mid;
      } else if (x.key < L[mid].key) {
         high = mid - 1;
      } else {
         low = mid + 1;
      }
   }
   return -1;
}
```

6.2.2 *Time Complexity*

Both iterative and recursive are of $O(\log(n))$

6.2.3 *Correctness*

Loop Invariant

- Two clauses to satisfy
  1. $1 \leqslant low \leqslant high \leqslant L.size$
  2. If our key is in L, then there is some target of the key where L[target] = key. Then, $1 \leqslant target \leqslant L.size$ and $low \leqslant target \leqslant high$, hence we are searching through the correct range for key
- ***Initialization***
  - Since our L is at least size 1, clause 1 holds
  - Suppose key is in L. This mean that $1 \leqslant target \leqslant L.size - 1$ holds true
- ***Maintenance***

- *BC 1*: $\mathtt{L}[\mathtt{mid}] < \mathtt{key}$
    - We search through elements in $\mathtt{L}[\mathtt{mid}+1, \mathtt{high}]$ since key is larger than mid
    - We know that key is $\leqslant \mathtt{high}$ by IH, otherwise it would violate the precondition
    - Hence, key is in $\mathtt{L}[\mathtt{mid}+1, \mathtt{high}]$
- *BC 2*: $\mathtt{L}[\mathtt{mid}] > \mathtt{key}$ (Similar to above case)
- *BC 3*: $\mathtt{L}[\mathtt{mid}] == \mathtt{key}$
    - Return mid, as key happened to be in the position mid was in
- *BC 1, 2, 3* holds true for all cases, thus we can conclude that LI holds for the $i+1$ iteration, as claimed

- ***Termination***
    - We must prove that if
        - $[\mathtt{low}_i, \mathtt{high}_i]$ on i^{th} iteration and
        - $[\mathtt{low}_{i+1}, \mathtt{high}_{i+1}]$ on iteration $i+1$,
    - $\mathtt{high}_{i+1} - \mathtt{low}_{i+1} < \mathtt{high}_i - \mathtt{low}_i$
    - By each condition, we will perform integer division
    - Hence, the array gets smaller on every iteration of the loop, eventually leading to termination

7 GRAPH TRAVERSAL

7.1 Depth-First Search

- Uses stack to explore a branch as far as possible before backtracking
- Backtracking achieved by popping from stack
- **IMPORTANT**: For this exam, push vertices onto the stack such that the vertex with the smallest ID (numerical value) is on the top
  - e.g. Adjacency List for Node 0: AL[0]→[1→2] or AL[0]→[2→1]
  - will be pushed onto the stack like the following:
  - (TOS) <1, 2> (BOS)

7.1.1 *Algorithm*

```
// Iterative
Queue<Integer> DFS(Graph g, int start) {
   Stack<Integer> stack = new Stack<>();
   Queue<Integer> queue = new Queue<>();
   Set<Integer> visited = new Set<>();

   stack.push(start);
   while (!stack.isEmpty()) {
      int u = stack.pop();
      visited.add(u);
      queue.enqueue(u);
      for (int v : G.adjacencyList[u]) {
         if (!visited.contains(v)) {
            stack.push(v);
         }
      }
   }
   return queue;
}
```

7.1.2 *Time Complexity*

7.1.3 *Correctness*

7.1.4 *Applications*

- **Cycle Detection**: Any time you encounter a new edge that points an already visited vertex then you have a cycle
- **Topological Sort**: applies to directed acyclic graph (no cycles)
  - Finds a valid order to execute tasks in a directed cyclic graph

7.2 Breadth-First Search

- Visit vertices by depth, meaning we visit every single vertex
- Uses a queue to explore each level before moving onto the next
- Backtracks by dequeueing from tmpQ
- **IMPORTANT**: For this exam, enqueue vertices into tmpQ such that vertices with smallest ID are towards the front of the queue

7.2.1 *Algorithm*

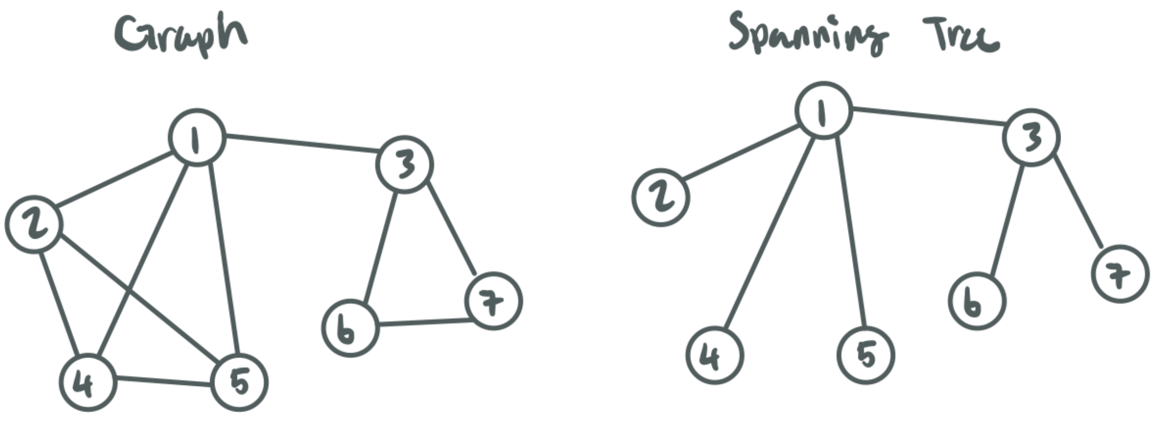
```
Queue<Integer> BFS(Graph g, int start) {
   Queue<Integer> queue = new Queue<>();
   Queue<Integer> tmpQ = new Queue<>();
   Set<Integer> visited = new Set<>();

   tmpQ.enqueue(start);
   while (!tmpQ.isEmpty()) {
      int u = tmpQ.dequeue();
      visited.add(u);
      queue.enqueue(u);
      for (int v : g.adjacencyList[u]) {
         if (!visited.contains(v)) {
            tmpQ.enqueue(v);
         }
      }
   }
   return queue;
}
```

7.2.2 *Time Complexity*

7.2.3 *Correctness*

7.2.4 *Applications*

Spanning Tree


Graph
Spanning Tree
1
2
3
4
5
6
7
1
2
3
4
5
6
7