Chapter I
Recursion

A function calling itself is known as recursion. It refers to a method of defining functions in which the function being defined is applied within its own definition, i.e., an algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task.

A function exhibits recursive behavior when it can be defined by two properties or rules:

1. A simple base case (or cases), and
2. A set of rules which reduce all other cases toward the base case called recursive case.

A recursive case has three components:
(a) Divide the problem into one or more simpler or smaller parts of the problems,
(b) Invoke the function (recursively) on each part, and
(c) Combine the solutions of the parts into a solution for the problem.

Example 1: The Factorial

We will begin with the example of the factorial, denoted “!”.

3! = 3 * 2 * 1

5! = 5 * 4 * 3 * 2 * 1

X! = X * (X - 1) * (X - 2) * (X - 3) * ... * 3 * 2 * 1

0! = 1

So how can we compute this!? Here is a recursive function that does it:
n! = (n - 1)! for n > 1, and n! = 1 for n ≤ 1. i.e.

- Factorial(0) is 1 [base case]
- Factorial(1) is 1 [base case]
- For all integers n > 1: n * factorial(n-1) [recursive definition]

The equivalent code in Java is

```java
int factorial(unsigned int n)
{
  if (n <= 1)
    return 1;
  else
    return n * factorial(n-1);
}
```

Example 2: The Fibonacci sequence
The Fibonacci sequence can be generated by a recursive function very similar to the recursive factorial function. The only difference is the rule, which for the Fibonacci sequence is $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ for $n > 1$, and $\text{fib}(n) = 1$ for $n \leq 1$. This kind of recursive definition appears frequently in mathematics, and is also called a recurrence relation.

- $\text{Fib}(0)$ is 0 [base case]
- $\text{Fib}(1)$ is 1 [base case]
- For all integers $n > 1$: $\text{Fib}(n)$ is $\text{Fib}(n-1) + \text{Fib}(n-2)$ [recursive definition]

It can be implemented in Java as

```java
unsigned int fib(unsigned int n)
{
    if (n == 0)
        return 0;
    elseif (n == 1)
        return 1;
    else
        return (fib(n-1)+ fib(n-2));
}
```

Example 3: The Power function

The Power function also is an example of recursion. The rule for power function is $\text{power}(x, n) = x \times \text{power}(x, n-1)$ for $n > 0$, and $\text{power}(x, n) = 1$ for $n = 0$.

```java
int power(double x, int n)
{
    if (n == 0)
        return 1;
    else
        return x * power(x,n-1);
}
```

**Method calls and Recursion Implementation**

**Activation Record**

A call stack is composed of stack frames (also called activation records or activation frames). These are machine dependent data structures containing subroutine state information. Each stack frame corresponds to a call to a subroutine which has not yet terminated with a return. The stack frame at the top of the stack is for the currently executing routine. The stack frame usually includes at least the following items:

- the arguments (Parameters and local variables) passed to the routine (if any)
- a dynamic link to the callers activation record
• the return address back to the routine's caller
• The return value (if any)

The block diagram of the same can be

![Activation record diagram]

**Anatomy of a recursive call**

It deals with how the recursive calls actually works. Take the example of power function, power(5.6,2). It starts with a main function, which calls the recursive function, power(5.6,2). An activation function for power(5.6,2)is created with 2,5.6 as local variables and return address to the main program say 136 and space for return value, which intern call power(5.6,1). power(5.6,1) again creates an activation record and calls power(5.6,0). Here the solution can be solved without any further recursion. The return value, 1 is stored and is the returned to power(5.6,1), which again returns 5.6 as return value to power(5.6,2) which finally returns 31.36 to the main function.
Tail recursion

In computer science, a **tail call** is a subroutine call that happens inside another procedure and that produces a return value, which is then immediately returned by the calling procedure. The call site is then said to be in **tail position**, i.e. at the end of the calling procedure. If a subroutine performs a tail call to itself, it is called **tail-recursive**. This is a special case of recursion. Tail calls are significant because they can be implemented without adding a new stack frame to the call stack. Most of the frame of the current procedure is not needed any more, and it can be replaced by the frame of the tail call, modified as appropriate.

Example

```c
void tail (int i)
{
    if ( i > 0 ) {
        printf (“%d”, i);
        tail (i-1);
    }
}
```

While writing tail recursion, it has to be noted that there should be only one recursive call, that should be the last statement and together with that call there should not be another call or operation.

Advantage of Tail Recursive Methods

1. Tail Recursive methods are easy to convert to iterative. For example
2. Smart compilers can detect tail recursion and convert it to iterative to optimize code.

3. It can be used to implement loops in languages that do not support loop structures explicitly.

**Non Tail recursion**

If a subroutine has to perform another call or operation after the recursive function, it is called non **tail-recursive**. For example:

```java
void nonTail (int i) {
    if (i > 0) {
        nonTail(i-1);
        System.out.print (i + "");
        nonTail(i-1);
    }
}
```

Here the function has to perform print and another call after the first recursive call and hence it is an example of non tail.

Another example is

```java
/* 200 */ void reverse() {
    char ch= getChar()
    if (ch != ‘n’) {
        reverse ();
        System.out.println(ch);
    }
}
```

Now the working of this algorithm is at first we are calling reverse() from mail() function, it inturns get a character ‘A’, then as is the property of recursive function it stores the address of the main function and printing of ‘A’ in stack. There is no need to store space for return value.
since the function is void. Then goes for the recursive call once again and gets character ‘B’. it once again stores the line to which it should returned and printing operation of ‘A’ in stack. This process is continued until ‘\n’ character is encountered. Now each of the operation to be performed is poped from the stack one by one and performed to get the result to be printed as ‘CBA’. The operation can be picturized as

The problem with non tailed algorithm is that it consumes lot of stack operations and is not efficient.

**Converting Non-tail to Tail Recursion**

A non-tail recursive method can be converted to a tail-recursive method by means of an "auxiliary" parameter used to form the result. The technique is usually used in conjunction with an "auxiliary" function. This is simply to keep the syntax clean and to hide the fact that auxiliary parameters are needed.

Example

```c
int fact_aux(int n, int result) {
    if (n == 1)
        return result;
    return fact_aux(n - 1, n * result)
}
int fact(n) {
    return fact_aux(n, 1);
}
```

**Excessive Recursion**

When recursive methods repeats the computations for some parameters, which results in long computation time even for simple cases. There may be possibility of repeating the already computed solutions. This types of recursion is called excessive recursion.
For example: Fibonacci Numbers

With this example we can easily show that how much the recursive formula is inefficient, let us try to see how Fib(6) is evaluated.

Here Fibonacci of 4 is computed twice, 3 is computed 3 times, 2 is computed 5 times, 1 is computed 8 times and for 0 is computed 5 times. This means that the same function is computed unnecessarily many times.

Backtracking

**Backtracking** is a general algorithm for finding all (or some) solutions to some computational problem that incrementally builds candidates to the solutions, and abandons each partial candidate $c$ ("backtracks") as soon as it determines that $c$ cannot possibly be completed to a valid solution. The classic textbook example of the use of backtracking is the eight queen’s puzzle that asks for all arrangements of eight queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of $k$ queens in the first $k$ rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned, since it cannot possibly be completed to a valid solution.

Before starting to eight queen problem, let us go for 4 queen problem

The solution to 4 queen problem is

```
  1  2  3  4
A  Q  Q  .
B  .  Q  Q
C  Q  .  Q
D  Q  Q  .
```

This can be achieved by backtracking as
As the definition of backtracking, we first place queen at position (1,1), then we go for row 2 here we cannot place the queen at (2,1) and (2,2). So we try to place the queen at (1,3). If we place the queen at (1,3) we cannot place the next queen at any of the position in next row and hence we backtrack the process from level 2 and place second queen at (2,4). This process is continued until we get the solution as given in the figure.

Similarly the solution to 8 queen problem is also can be achieved as

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td>Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Q</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The algorithm for the same is
Hence Backtracking is a method of problem solving by well organized trial and error. We should make sure that we never try the same thing twice.

```
putQueen(row)
    for every position col on the same row
        if position col is available
            place the next queen in position col;
        if (row < 8)
            putQueen(row+1);
        else success;
        remove the queen from position col;
```
Chapter II

Trees

It is an acyclic connected graph where each node has zero or more children nodes and at most one parent node. A node is a structure which contains a value and addresses of the left and right children. Each node in a tree has zero or more child nodes, which are below it in the tree. A node that has a child is called the child's parent node. A node has at most one parent. An internal node or inner node is any node of a tree that has child nodes. Similarly, an external node (also known as an outer node, leaf node, or terminal node), is any node that does not have child nodes. The topmost node in a tree is called the root node. Examples of trees are

Binary Trees

A binary tree is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right". Nodes with children are parent nodes. Examples of binary trees are

Binary Search Tree (BST)

It is a binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.
Example for BST is

![BST Example](image)

**Implementation of BST**

The BST can be implemented in Java as a collection of nodes. Each node contains key value and addresses for its left and right children. It can be implemented as

```java
class Node {
    int key; // data item (key)
    Node left; // this node's left child
    Node right; // this node's right child

    public Node (){
        left=null; right=null;
    }

    public Node (int val){
        this(val,null,null);
    }

    public Node (int val, Node lt, Node rt){
        key=val; left=lt; right=rt;
    }

    public void visit (int val){
        System.out.println(key + "");
    }
}
```

Here we have three variable members key, left and right which are of types integer, node and node respectively. We also have constructors to initialise nodes. The class Node also has a public method visit to print current key value.
**Searching in BST**

Searching a binary search tree for a specific value can be a recursive process. We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left subtree. Similarly, if it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.

For example suppose we want to find the element 4, then we start at the root, the key value is 8 which is greater than 5. So we traverse to its left child, here the value is 3 which is less than 5. So we traverse right, here the key value is 6 which is greater than 5. Now we traverse left and the value is 5 which is equal to the element to be searched. 5. We stop the procedure and conclude that 5 is present in this BST.

The java code for searching an element in a BST is

```java
public IntBSTNode search(IntBSTNode p, int el) {
    while (p != null) {
        if (el == p.key)
            return p;
        else if (el < p.key)
            p = p.left;
        else p = p.right;
    }
    return null;
}
```

**Tree Traversal**

Tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node), traversing to the left child node, and traversing to the right child node. Thus the process is most easily described through recursion. The methods traversing a tree are Breadth First Search and Depth First search.
Breadth First Search (BFS)

Trees can also be traversed in **level-order**, where we visit every node on a level before going to a lower level. This is also called Breadth-first traversal. For example, the tree shown below can be traversed in BFS method as

![Tree Diagram](image)

First the level zero is considered and visited, next level1 is considered and visited one by one. This process is continued until we reach the last node. Ie, the traversal results in printing the nodes as, F, B, G, A, D, I, C, E, H. The algorithm for the same is

```java
public void breadthFirst() {
    IntBSTNode p = root;
    Queue queue = new Queue;
    if (p != null) {
        queue.enqueue(p);
        while (!queue.isEmpty()) {
            p = (IntBSTNode) queue.dequeue();
            p.visit;
            if (p.left != null)
                queue.enqueue(p.left);
            if (p.right != null)
                queue.enqueue(p.right);
        }
    }
}
```

Depth First search

**Depth-first search (DFS)** is an algorithm for traversing or searching a **tree**. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch. The methods for doing DFS are preorder, inorder and postorder. Let us consider an expression tree
Preorder Traversal
To traverse a non-empty binary tree in **preorder**, perform the following operations recursively at each node, starting with the root node:

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

The output for preorder traversal is `+/ab*-cde`

Inorder Traversal

To traverse a non-empty binary tree in **inorder** (**symmetric**), perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

The output for inorder traversal is `a/b+c-d*e`

Postorder Traversal

To traverse a non-empty binary tree in **postorder**, perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

The output for inorder traversal is `ab/cd-e*+`

The java code for the preorder, in order and post order traversals in a tree are
Insertion in BST

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

Algorithm for insertion
For inserting a node, its value is first compared with the value of the root. If its value is less than the root's, it is then compared with the value of the root's left child. If its value is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value. The algorithm is as follows:

```java
public void insert(int el) {
    IntBSTNode p = root, prev = null;
    while (p != null) { // find a place for inserting new node;
        prev = p;
        if (p.key < el)
            p = p.right;
        else p = p.left;
    }
    if (root == null) // tree is empty;
        root = new IntBSTNode(el);
    else if (prev.key < el)
        prev.right = new IntBSTNode(el);
    else prev.left = new IntBSTNode(el);
}
```

**Deletion in BST**

There are three possible cases to consider:

- **Deleting a leaf (node with no children):** Deleting a leaf is easy, as we can simply remove it from the tree.
- **Deleting a node with one child:** Remove the node and replace it with its child.
- **Deleting a node with two children:** Call the node to be deleted $N$. Do not delete $N$. Instead, choose either its in-order successor node (minimum value in the right subtree) or its in-order predecessor node (maximum value in the left subtree), $R$. Replace the value of $N$ with the value of $R$, then delete $R$.

As with all binary trees, a node's in-order successor is the left-most child of its right subtree (minimum value in the left subtree), and a node's in-order predecessor is the right-most child of its left subtree (maximum value in the left subtree). In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

**Example:**
Deleting a node with two children from a binary search tree. The triangles represent subtrees of arbitrary size, each with its leftmost and rightmost child nodes at the bottom two vertices.

Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an unbalanced tree, so good implementations add inconsistency to this selection.
AVL Trees

AVL tree is a self-balancing binary search tree where the heights of the two child subtrees of any node differ by at most one. The balance factor of a node is the height of its left subtree minus the height of its right subtree (sometimes opposite) and a node with balance factor 1, 0, or −1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.

Insertion

Pictorial description of how rotations cause rebalancing tree, and then retracing one's steps toward the root updating the balance factor of the nodes. The numbered circles represent the nodes being balanced. The lettered triangles represent subtrees which are themselves balanced BSTs.

After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains −1, 0, or +1 then no rotations are necessary. However, if the balance factor becomes ±2 then the subtree rooted at this node is unbalanced. If insertions are performed serially, after each insertion, at most one of the following cases needs to be resolved to restore the entire tree to the rules of AVL.

There are four cases which need to be considered, of which two are symmetric to the other two. Let P be the root of the unbalanced subtree, with R and L denoting the right and left children of P respectively.

Right-Right case and Right-Left case:

- If the balance factor of P is -2 then the right subtree outweights the left subtree of the given node, and the balance factor of the right child (R) must be checked. The left rotation with P as the root is necessary.
- If the balance factor of R is -1, a single left rotation (with P as the root) is needed (Right-Right case).
- If the balance factor of R is +1, two different rotations are needed. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root (Right-Left case).

Left-Left case and Left-Right case:

- If the balance factor of P is +2, then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must be checked. The right rotation with P as the root is necessary.
- If the balance factor of L is +1, a single right rotation (with P as the root) is needed (Left-Left case).
If the balance factor of L is -1, two different rotations are needed. The first rotation is a left rotation with L as the root. The second is a right rotation with P as the root (Left-Right case).

Deletion

If the node is a leaf or has only one child, remove it. Otherwise, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as a replacement has at most one subtree. After deletion, retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.
As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

In addition to the balancing described above for insertions, if the balance factor for the tree is 2 and that of the left subtree is 0, a right rotation must be performed on P. The mirror of this case is also necessary.

The retracing can stop if the balance factor becomes −1 or +1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes −2 or +2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

**Priority Queues**

A priority queue is essentially a list of items in which each item has associated with it a *priority*. In general, different items may have different priorities and we speak of one item having a higher priority than another. Given such a list we can determine which is the highest (or the lowest) priority item in the list. Items are inserted into a priority queue in any, arbitrary order. However, items are withdrawn from a priority queue in order of their priorities starting with the highest priority item first. Using a Binary heap one can implement priority queue.

**Binary heap**

A *binary heap* is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints:

- The *Structure property*: the tree is a complete binary tree; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- The heap property: each node is greater than or equal to each of its children in the case of max heap. For a min heap, each node is less than or equal to each of its children.

Example of a complete binary max heap

![Example of a complete binary max heap](image)

Example of a complete binary min heap

![Example of a complete binary min heap](image)

**Heap operations**

**Insert**

To add an element to a heap we must perform an *up-heap* operation (also known as *bubble-up*, *percolate-up*, *sift-up*, *trickle up*, *heapify-up*, or *cascade-up*) in order to restore the heap property.

1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

Say we have a max-heap

![Max-heap diagram](image)

and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 is greater than 8, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:

![Max-heap after first swap](image)

However the heap property is still violated since 15 is greater than 11, so we need to swap again:
which is a valid max-heap. There is no need to check the children after this. Before we placed 15 on X, the heap was valid, meaning 11 is greater than 5. If 15 is greater than 11, and 11 is greater than 5, then 15 must be greater than 5. As the case of max heap, the insertion operation can be done for min heap also. Here the only difference is we swap the root and child only when child is less.

**Remove**

The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *sift-down*, *trickle down*, *heapify-down*, or *cascade-down*).

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

So, if we have the same max-heap as before, we remove the 11 and replace it with the 4.

![Max-heap before removal](image)

Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements 4 and 8, is enough to restore the heap property and we need not swap elements further:

![Max-heap after removal](image)

The downward-moving node is swapped with the *larger* of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position.
Multiway Trees

A B-tree of order $m$ (the maximum number of children for each node) is a tree which satisfies the following properties:

1. Every node $x$ has the following fields:
   a. $n[x]$, the number of keys currently stored in node $x$,
   b. the $n[x]$ keys themselves, stored in nondecreasing order, so that $key_1[x] \leq key_2[x] \leq \cdots \leq key_{n[x]}[x]$,
   c. $leaf[x]$, a boolean value that is $true$ if $x$ is a leaf and $false$ if $x$ is an internal node.

2. Each internal node $x$ also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their $c_i$ fields are undefined.

3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if $k_i$ is any key stored in the subtree with root $c_i[x]$, then $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \cdots \leq key_{n[x]}[x] \leq k_{n[x]+1}$.

4. All leaves have the same depth, which is the tree’s height $h$.

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree:
   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least $t$ children. If the tree is nonempty, the root must have at least one key.
   b. Every node can contain at most $2t - 1$ keys. Therefore, an internal node can have at most $2t$ children. We say that a node is full if it contains exactly $2t - 1$ keys.1
   The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree. In practice, however, much larger values of $t$ are typically used.

Example

![Diagram of a B-tree]

<table>
<thead>
<tr>
<th>Depth of Node</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$2t$</td>
</tr>
<tr>
<td>3</td>
<td>$2t^2$</td>
</tr>
</tbody>
</table>

Splitting a Node
Insertion

(b) \( B \) inserted

(c) \( Q \) inserted

(d) \( L \) inserted

(e) \( F \) inserted

Deleting a node from B-tree

1. If the key \( k \) is in node \( x \) and \( x \) is a leaf, delete the key \( k \) from \( x \).
2. If the key \( k \) is in node \( x \) and \( x \) is an internal node, do the following.
a. If the child $y$ that precedes $k$ in node $x$ has at least $t$ keys, then find the predecessor $k_-$ of $k$ in the subtree rooted at $y$. Recursively delete $k_-$, and replace $k$ by $k_-$ in $x$. (Finding $k_-$ and deleting it can be performed in a single downward pass.)

b. Symmetrically, if the child $z$ that follows $k$ in node $x$ has at least $t$ keys, then find the successor $k_+$ of $k$ in the subtree rooted at $z$. Recursively delete $k_+$, and replace $k$ by $k_+$ in $x$. (Finding $k_+$ and deleting it can be performed in a single downward pass.)

c. Otherwise, if both $y$ and $z$ have only $t-1$ keys, merge $k$ and all of $z$ into $y$, so that $x$ loses both $k$ and the pointer to $z$, and $y$ now contains $2t-1$ keys. Then, free $z$ and recursively delete $k$ from $y$.

3. If the key $k$ is not present in internal node $x$, determine the root $c_i[x]$ of the appropriate subtree that must contain $k$, if $k$ is in the tree at all. If $c_i[x]$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least $t$ keys. Then, finish by recursing on the appropriate child of $x$.

a. If $c_i[x]$ has only $t-1$ keys but has an immediate sibling with at least $t$ keys, give $c_i[x]$ an extra key by moving a key from $x$ down into $c_i[x]$, moving a key from $c_i[x]$’s immediate left or right sibling up into $x$, and moving the appropriate child pointer from the sibling into $c_i[x]$.

b. If $c_i[x]$ and both of $c_i[x]$’s immediate siblings have $t-1$ keys, merge $c_i[x]$ with one sibling, which involves moving a key from $x$ down into the new merged node to become the median key for that node.
Graph Theory

Graph

Definition 1 (Graph) A graph $G$ is a pair $G = (V,E)$ where
- $V$ is a finite set, called the vertices of $G$, and
- $E$ is a subset of $\mathcal{P}_2(V)$ (i.e., a set $E$ of two-element subsets of $V$), called the edges of $G$.

Example 1:

![Graph Example](image)

Definition 2 (Loops) A loop is an edge that connects a vertex to itself

Example 2:

![Loop Examples](image)

Definition 3 (Degrees of vertices) the **degree** (or **valency**) of a vertex of a graph is the number of edges incident to the vertex, with loops counted twice. The degree of a vertex $v$ is denoted $\deg(v)$. The maximum degree of a graph $G$, denoted by $\Delta(G)$, and the minimum degree of a graph, denoted by $\delta(G)$, are the maximum and minimum degree of its vertices. In the graph in example 1, the maximum degree is 3 and the minimum degree is 1.

Definition 4 (Path) A **path** in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence. Another definition is the sequence of edges $e_1, e_2, \ldots, e_{n-1}$ is called a path. A path may be infinite, but a finite path always has a first vertex, called its **start vertex**, and a last vertex, called its **end vertex**. Both of them are called **terminal vertices** of the path. The other vertices in the path are **internal vertices**. A cycle is a path such that the start vertex and end vertex are the same.
Definition 5 (Connected Graph) A graph is said to be connected when there exist at least one path between every pair of vertices.

Definition 6 (Directed graph) A directed graph or digraph is a pair $G = (V, E)$ where $V$ is a set of elements called vertices or nodes and $E$ is a set of ordered pairs of vertices, called arcs, directed edges, or arrows. An in degree of a node in a directed graph is the number of edges coming in to it and An out degree of a node in a directed graph is the number of edges going from it.

Example:

![Directed Graph Example](image)

Definition 7 (weakly connected) A digraph $G$ is called weakly connected if the undirected underlying graph obtained by replacing all directed edges of $G$ with undirected edges is a connected graph. A digraph is strongly connected or strong if it contains a directed path from $u$ to $v$ and a directed path from $v$ to $u$ for every pair of vertices $u, v$. The strong components are the maximal strongly connected subgraphs.

Example

![Strong Components Example](image)

A directed acyclic graph or acyclic digraph is a directed graph with no directed cycles.

Example

![Directed Acyclic Graph Example](image)

subgraph of a graph $G$ is a graph whose vertex set is a subset of that of $G$, and whose adjacency relation is a subset of that of $G$ restricted to this subset. In the other direction, a supergraph of a graph $G$ is a graph of which $G$ is a subgraph.

A weighted graph associates a label (weight) with every edge in the graph. Weights are usually real numbers. They may be restricted to rational numbers or integers. The weight of a path or
the **weight of a tree** in a weighted graph is the sum of the weights of the selected edges. Sometimes a non-edge is labeled by a special weight representing infinity. Sometimes the word **cost** is used instead of weight. When stated without any qualification, a graph is always assumed to be unweighted. In some writing on graph theory the term **network** is a synonym for a **weighted graph**. A network may be directed or undirected, it may contain special vertices (nodes), such as **source** or **sink**.

Example:

![Graph Example](image)

**Trees**

A **tree** is an undirected graph in which any two vertices are connected by **exactly one** simple path. In other words, any connected graph without cycles is a tree. A **forest** is a disjoint union of trees.

Examples

![Forest Examples](image)

**Spanning Tree**

A **spanning tree** $T$ of a connected, undirected graph $G$ is a tree composed of all the vertices and some (or perhaps all) of the edges of $G$. Informally, a spanning tree of $G$ is a selection of edges of $G$ that form a tree **spanning** every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed.

**Minimum Spanning Tree**

A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree.

**Kruskal's algorithm**

**Kruskal's algorithm** is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that
includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest*.

The Algorithm

Step 1: find out the edge with the next minimum weight in the graph

Step 2: remove this edge from the graph and add this edge to the tree if it does not form a cycle. otherwise discard that edge

Step 3: if tie occurs then arbitrarily select one. Repeat the process step1 and 2 until all vertices are connected.

At the termination of the algorithm, Resultant Tree is the minimum spanning tree forms the minimum spanning tree of the graph

Example:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

AD and CE are the shortest arcs, with length 5, and AD has been arbitrarily chosen, so it is highlighted.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

CE is now the shortest arc that does not form a cycle, with length 5, so it is highlighted as the second arc.
The next arc, DF with length 6, is highlighted using much the same method.

The next-shortest arcs are AB and BE, both with length 7. AB is chosen arbitrarily, and is highlighted. The arc BD has been highlighted in red, because there already exists a path (in green) between B and D, so it would form a cycle (ABD) if it were chosen.

The process continues to highlight the next-smallest arc, BE with length 7. Many more arcs are highlighted in red at this stage: BC because it would form the loop BCE, DE because it would form the loop DEBA, and FE because it would form FEBAD.

Finally, the process finishes with the arc EG of length 9, and the minimum spanning tree is found.
Graph Representation

(a) Graph

(b) Adjacency List

(c) Adjacency Matrix

(d) Adjacency List

(e) Adjacency Matrix
**Graph traversal** refers to the problem of visiting all the nodes in a graph in a particular manner. Tree traversal is a special case of graph traversal. In contrast to tree traversal, in general graph traversal, each node may have to be visited more than once, and a root-like node that connects to all other nodes might not exist.

**Depth-first Search**

A Depth-first search (DFS) is a technique for traversing a finite undirected graph. DFS traverses the depth of the tree before the breadth. That called DFS.

**Algorithm**

Step 1: Visit the first Vertex

Step 2. Visit the first child while keeping all other children in stack

Step 3: do step 2 until all nodes are visited

**Example**

First we visit node A then we visit E while keeping D and F. Like that final order is A, E, C, B, D, F.

**Breadth-first Search**

A Breadth-first search (BFS) is another technique for traversing a finite undirected graph. BFS visits the sibling nodes before visiting the child nodes.
Algorithm

Step 1: Visit the first Vertex

Step 2. Visit all it’s children

Step 3: take the next child and do step 2 until all nodes are visited

Example

![Graph Image]

First we visit node A then we visit E, D, F, then take F visit all it’s children B. Like that final order is A, E, D, F, B, C.

Chapter 5
Hashing

A hash function is any algorithm or subroutine that maps large data sets, called keys, to smaller data sets. For example, a single integer can serve as an index to an array. The values returned by a hash function are called hash value. Hash table is a data structure that uses a hash function to map identifying values.

Hash functions are mostly used to accelerate table lookup tasks such as finding items in a database.

If a hash function maps two or more keys to the same hash value, then the problem is called collision.

Different Hash Functions

1. The Division Method
Map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

\[ h(k) = k \mod m. \]

Example: If table size m = 12

key k = 100 then \( h(100) = 100 \mod 12 = 4 \)

2. Mid-Square Method

In this algorithm you square the key and then select certain bits. Usually the middle half of the bits is taken. The mixing provided by the multiplication ensures that all digits are used in the computation of the hash code.

Example: Let the keys range between 1 and 32000 and let the TableSize be 2048 = 2^{11}.

Square the Key and remove the middle 11 bits. (Grabbing certain bits of a word is easy to do using shift operators in assembly language or can be done using the div and mod operators using powers of two.)

In general r bits gives a table of size \( 2^r \).

3. Folding Method

Break the key into pieces (sometimes reversing alternating chunks) and add them up.

This is often used if the key is too big. E.g., If the keys are Social security numbers, the 9 digits will generally not fit into an integer. Break it up into three pieces - the 1st digit, the next 4, and then the last 4. Then add them together. Now you can do arithmetic on them. Then apply the hashing function on it

4. Extraction Method

In extraction method, only a part of the key is used to compute the address. For example, in the case of social security number 123-65-456, it may take the first 2 digits together with the last two digits. Here it is 1256.

5. Radix Transformation

In radix transformation the key is transformed in to another number base. For example, if the key is in decimal number system then its value is transformed into octal number and applies the transformation.

Collision Resolution
It is a process of resolving collisions in a hash table. The main methods of doing the same are open addressing and chaining

**Open Addressing**

Open addressing, or closed hashing, is a method of collision resolution in hash tables. With this method a hash collision is resolved by probing, or searching through alternate locations in the array until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. The methods are

1. **Linear probing:** first hash the key with any hashing function. Then if a collision occurs then put the key in the next available position. For example if we use if the hashing function is $mod\ 10$ and the size of the hash table is 10 then the insertion of different key values can be done by

   ![Linear Probing Example](image)

2. **Quadratic probing:** first hash the key with any hashing function. Then if a collision occurs then put the key in the next available $i^2$ position. For example if we use if the hashing function is $mod\ 10$ and the size of the hash table is 10 then the insertion of different key values can be done by

   ![Quadratic Probing Example](image)

2. **Double Hashing:** first hash the key with any hashing function. Then if a collision occurs then put the key according to the second hash function. For example if we use first hashing function
as \( mod \, 10 \), second hashing function as \( \text{hash2}(x) = R - (x \mod R) \) where \( R \) is a prime number smaller than hash table size and the size of the hash table is 10 then the insertion of different key values can be done by

<table>
<thead>
<tr>
<th>Empty Table</th>
<th>After 89</th>
<th>After 18</th>
<th>After 49</th>
<th>After 58</th>
<th>After 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>58</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>49</td>
<td>49</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
</tr>
</tbody>
</table>

The first collision occurs when 49 is inserted. \( h2(49) = 7 - 0 = 7 \), so 49 is inserted in position 6. \( h2(58) = 7 - 2 = 5 \), so 58 is inserted at location 3. Finally, 69 collides and is inserted at a distance \( h2(69) = 7 - 6 = 1 \) away. If we tried to insert 60 in position 0, we would have a collision. Since \( h2(60) = 7 - 4 = 3 \), we would then try positions 3, 6, 9, and then 2 until an empty spot is found.

**Chaining**

It is a type of collision resolution method which insert the element in separate linked list according to the hash function. Here each slot of the bucket array is a pointer to a linked list that contains the key-value pairs that hashed to the same location. Lookup requires scanning the list for an entry with the given key. For example if we use if the hashing function is \( mod \, 10 \) and the size of the hash table is 10 then the insertion of different key values can be done by