Hashing

- Search Time:
  - Linear : $O(n)$
  - Non-Linear : $O(log(n))$
- Hashing Allows searches in $O(1)$ time.
- Changes Keys to a shorter value that makes it easier to find original key.
- Used in : Hash Tables
- Hash Function : maps keys into hash table range
  - Folding
  - Truncation
  - Modulo
- Collision : multiple keys generate same hash
  - Open addressing(Closed Hashing)
    - Every node of the hashtable has a `data` and `flag`. The flag indicates if the cell is occupied.
    - Linear Probing
      - $h(key) = (h(key) + i)\%Tabsize$
      - finds the next vacant spot
      - Collisions encountered while resolving a collision are not counted as collisions.

```
void insert(int key,NODE *hashTable){
    int hash,i=0;
    hash=((key%SIZE)+i)%SIZE;
    while(hashTable[hash].flag==1 && i<SIZE){
        i++;
        hash=((key%SIZE)+i)%SIZE;
    }
    if(hashTable[hash].flag==0){
        hashTable[hash].data=key;
        hashTable[hash].flag=1;
    }
    else
        printf("\nData cannot be inserted");
}
```

- Quadratic Probing
  - $h(key) = (h(key) + i^2)\%Tabsize$
- Double Hashing
  - $h(key) = (h_1(key) + i * h_2(key))\%Tabsize$

```
void insert(int key,NODE *hashTable){
    int hash,hash2,i=0;
    hash=((key%SIZE)+i*hash2)%SIZE;
    hash2=7-(key%7);
    while(hashTable[hash].flag==1 && i<SIZE){
        i++;
        hash=((key%SIZE)+i*hash2)%SIZE;
    }
    if(hashTable[hash].flag==0){
        hashTable[hash].data=key;
        hashTable[hash].flag=1;
    }
    else
        printf("\nData cannot be inserted");
}
```

- Separate Chaining (Open Hashing)
  - Every cell points to a linked list

```
void insert(int key,HASHTABLE *HashTable){
    int hash;
    NODE *nn,*t;
    hash=key%SIZE;
    nn=createNode(key);
    if(HashTable[hash].count==0){
       HashTable[hash].count++;
       HashTable[hash].head=nn;
    }
    else{
      t=HashTable[hash].head;
      while(t->next!=NULL)
            t=t->next;
      t->next=nn;
```

```
        HashTable[hash].count++;
    }
}
```

- Rehashing

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    int flag;
};
typedef struct {
    int size;
    struct node *hashTable;
} HASH;

int count = 0;

HASH *createHash(int size) {
    HASH *hash = (HASH *)malloc(sizeof(HASH));
    hash->size = size;
    hash->hashTable=(struct node*)malloc(size*sizeof(struct node));
    return hash;
}

void destroyHash(HASH *hash) {
    free(hash->hashTable);
    free(hash);
}

void rehash(int key, HASH **h);

void insert_(int key, HASH *h) {
    int hash;
    int i = 0;
    count++;
    printf("\ncount=%d",count);
    if (count > (float)(0.75 * h->size)) {
        printf("\n***");
        rehash(key, &h);
    } else {
        hash = ((key % h->size) + i) % h->size;
        while (h->hashTable[hash].flag != 0 && i < h->size) {
            i++;
            hash=((key % h->size)+i) % h->size;
        }
        if (h->hashTable[hash].flag == 0) {
            h->hashTable[hash].data = key;
            h->hashTable[hash].flag = 1;
            printf("The data %d is inserted at %d\n", key, hash);
        } else
            printf("\nData cannot be inserted");
    }
}

void rehash(int key, HASH **h) {
    int oldSize = (*h)->size;
    struct node *oldTable = (*h)->hashTable;

    (*h)->size = 2 * oldSize;
    (*h)->hashTable = (struct node *)calloc((*h)->size, sizeof(struct node));
    count = 0;

    for (int i = 0; i < oldSize; i++) {
        if (oldTable[i].flag == 1) {
            insert_(oldTable[i].data, *h);
        }
    }
    insert_(key, *h);
    free(oldTable);
}
void display(HASH *h) {
    printf("\nHash Table size:%d\n", h->size);
    for (int i = 0; i < h->size; i++) {
        if (h->hashTable[i].flag == 1) {
            printf("%d %d\n", i, h->hashTable[i].data);
        }
    }
}
```

Trie

- Each node contains `m` pointers for `m` possible symbols at each position.
- Each node has an `end of word` flag to denote end of words

Applications:

- English dictionary
- Predictive text
- Auto-complete dictionary found on Mobile phones and other gadgets.

Advantages:

- Faster than BST
- Printing of all the strings in the alphabetical order easily.
- Prefix search can be done (Auto complete).

Disadvantages:

- Need for a lot of memory to store the strings,
- Storing of too many node pointers.

```
struct trie{
    struct trie *child[255];
    int eos;
};
struct stack{
    struct trie *node;
    int index;
};
```

Add words

```
void insertWord(char *word,Trie *root)
{
    int index;
    Trie *t=root;
    for(int i=0;word[i]!='\0';i++)
    {
        index=word[i];//word[i]-'a' - 26 pointers
        if(t->child[index]==NULL)
            t->child[index]=createNode();
        t=t->child[index];
    }
    t->eos=1;
}
```

Display

```
void display(Trie *root){
    Trie *t=root;
    for(int i=0;i<255;i++)
    {
        if(t->child[i]!=NULL)
        {
            output[len++]=i;//i+'a' =>26 pointers
            if(t->child[i]->eos==1)
            {
                printf("\n");
                for(int j=0;j<len;j++)
                    printf("%c",output[j]);
            }
            display(t->child[i]);
        }
    }
    len--;
    return;
}
```

Search

```
void search(char *word,Trie *root){
    int index;
    Trie *t=root;
    for(int i=0;word[i]!='\0';i++){
        index=word[i];
        if(t->child[index]==NULL){
            printf("\nData not found");
```

```
            return;
        }
        t=t->child[index];
    }
    if(t->eos==1)
        printf("\nData found");
    else
        printf("\nData not found");
    return;
}
```

Deletion

```
void deleteData(char *word,Trie *root){
    int index;
    Stack *s;
    Trie *t=root;
    for(int i=0;word[i]!='\0';i++){
        index=word[i];
        if(t->child[index]==NULL){
            printf("\nData not found");
            return;
        }
        push(t,index);
        t=t->child[index];
    }
    t->eos=0;
    if(ChildCount(t)>=1)
        return;
    else{
        s=pop();
        t=s->node;
        index=s->index;
        while(ChildCount(t)<=1 && t->eos==0){
            free(t->child[index]);
            t->child[index]=NULL;
            s=pop();
            t=s->node;
            index=s->index;
        }
    }
}
```

Graphs

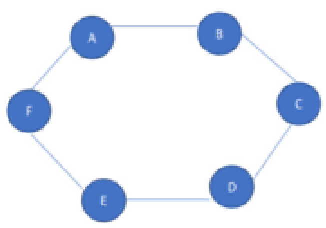
Application of DFS

- Detecting whether a cycle exist in graph.
- Finding a path in a network
- Topological Sorting: Used for job scheduling
- To check whether a graph is strongly connected or not

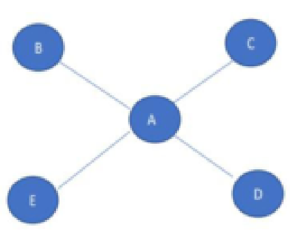
Application of BFS

- Finding the shortest path
- Social Networking websites like twitter, Facebook etc.
- GPS Navigation system
- Web crawlers
- Finding a path in network
- In Networking to broadcast the packets

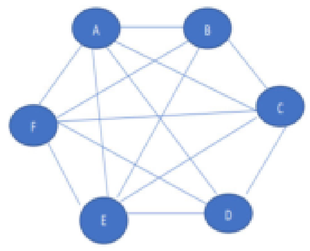
- Connected graph : there is a path between any pair of vertices
- there is no unreachable vertex
- Topology : order of arranging the nodes
  - Ring



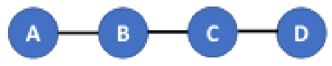
  - Star



- Mesh



- Bus



BFS Implementation

```
1 ---- 2
|      |
|      |
4 ---- 3
```

- 2 Ways :
- Adjacency list

```
        1 -> 2 -> 4
        2 -> 1 -> 3
        3 -> 2 -> 4
        4 -> 1 -> 3`
```

- Adjacency Matrix

```
           1  2  3  4
        1  0  1  0  1
        2  1  0  1  0
        3  0  1  0  1
        4  1  0  1  0
```

```
// Creating Adjacency List
int create_adjList(NODE *l){
    int v,from,to;
    NODE *t,*nn;
    printf("\nEnter the no. of nodes:");
    scanf("%d",&v);
    l[0].data=v; // first node stores number of elements
    l[0].next=NULL;
    for(int i=1;i<=l[0].data;i++){
        l[i].data=i; // filling up adjacency list
        l[i].next=NULL;
    }
    while(1){
    printf("\nEnter the from and to node");
    scanf("%d %d",&from,&to);
    if(from>0 && from<=v && to>0 && to<=v){
            t=&l[from]; // pass by reference
            while(t->next!=NULL){
                t=t->next;
            }
            nn=(NODE*)malloc(sizeof(NODE));
            nn->data=to;
            nn->next=NULL;
            t->next=nn;
    }
    else
        break;
    }
     for(int i=1;i<=v;i++)
    {
        if(!visited[i])
            return 0;
    }
return 1;
}
```

```
// Checking for connectivity using bfs with adjacency list
 void bfs_AjdList(NODE *l)
 {
     int source,*queue,*visited,v,i,j;
     NODE *t;
     printf("\nEnter the source vertex");
```

```
    scanf("%d",&source);
    v=l[0].data; // number of nodes in the graph
    queue=(int*)calloc(v,sizeof(int));
    visited=(int*)calloc(v+1,sizeof(int));
    enqueue(queue,source); // add visited nodes to queue
    visited[source]=1; // mark visited nodes
    printf("\n%d ",source);
    while(!isempty(queue))
    {
        i=dequeue(queue);
        t=&l[i];
        while(t->next!=NULL)
        {
           t=t->next;
           j=t->data;
           if(visited[j]==0){
               enqueue(queue,j);
               visited[j]=1;
               printf("%d ",j);
           }
        }
    }
 }
```

```
// Creating an Adjacency Matrix
void create_adjMatrix(GRAPH *g){
    int from,to;
    printf("\nEnter the no. of Vertices:");
    scanf("%d",&g->vertex);
    for(int i=1;i<=g->vertex;i++){
        g->adjMatrix[0][i]=i;
        g->adjMatrix[i][0]=i;
        for(int j=1;j<=g->vertex;j++){
            g->adjMatrix[i][j]=0;
        }
    }
    while(1){
      printf("\nEnter the from and to vertices:");
      scanf("%d %d",&from,&to);
      if(from>0 && from<=g->vertex && to>0 && to<=g->vertex){
        g->adjMatrix[from][to]=1;
        //g->adjMatrix[to][from]=1; for undirected
        }
      else
        break;
    }
}
```

```
// BFS with Adjacency Matrix
int bfs_AdjMat(GRAPH *g)
 {
     int source,*queue,*visited,v,i;
     printf("\nEnter the source vertex");
     scanf("%d",&source);
     v=g->vertex;
     queue=(int*)calloc(v,sizeof(int));
     visited=(int*)calloc(v+1,sizeof(int));
     enqueue(queue,source);
     visited[source]=1;
     printf("\n%d ",source);
     while(!isempty(queue)){
         i=dequeue(queue);
         for(int j=1;j<=v;j++){
            if(g->adjMatrix[i][j]==1 && visited[j]==0){
                enqueue(queue,j);
                visited[j]=1;
                printf("%d ",j);
            }
         }
     }
     for(int i=1;i<=g->vertex;i++)
    {
        if(!visited[i])
            return 0;
    }
return 1;
 }
```

```
// DFS using Adjacency List
void dfs_adjList(NODE *l,int source){
```

```
    NODE *t;
    static int visited[MAX];
    t=&l[source];
    printf("%d->",t->data);
    visited[source]=1;
    while(t->next!=NULL){
        t=t->next;
        if(visited[t->data]==0)//if unvisited
            dfs_adjList(l,t->data);
        }
    }
```

```
// DFS using adjacency matrix
void dfs_adjMat(GRAPH *g,int start,int parent){
    static int visited[MAX];
    printf("%d ",start);
    visited[start]=1;
    for(int i=1;i<=g->vertex;i++){
        if(g->adjMatrix[start][i]==1){
            if(visited[i]==0){
                dfs_adjMat(g,i,start);
            }
            else if(i!=parent){
                printf("\nCycle detected: %d -> %d\n", start, i);
            }
        }

    }
}
```

Btree

- All leaves at same level
- Left subtree has lesser nodes than right subtree
- For n degree B-tree
  - Max number of key values $m-1$
  - Except root all nodes but contain minimum $ceil(m/2)-1$ keys.
  - Max number of child nodes is m
  - Minimum number of children a node can have $ceil(m/2)$.

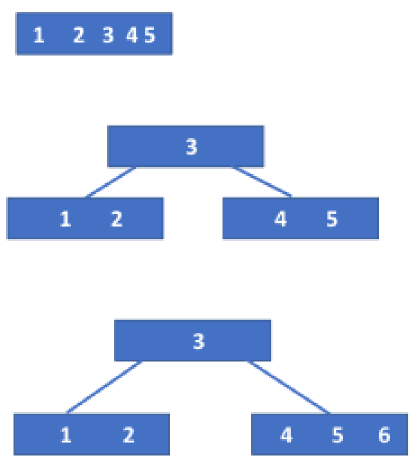
Why use B-Tree

- B-tree reduces the number of reads made on the disk
- B Trees can be easily optimized to adjust its size according to the disk size
- It is a specially designed technique for handling a bulky amount of data.
- It is a useful algorithm for databases and file systems.
- B-tree is efficient for reading and writing from large blocks of data

Insertion

- If leaf node has fewer than $m-1$ keys, then insertion continues into the same node.
- If this is false, node is split with median of the node as the parent and 2 children.
- Insertion takes place only on the leaf.

Insert 6:



Btree of degree 5

Deletion

- Always done on leaf node
- If node has more than critical number of nodes then delete the key
- Otherwise:
  - if left node has more than critical number of keys then push the largest element to parent and push the intervening element down to the right node.
  - If right node has more than critical number of keys then push the largest element to parent and push the intervening element down tot he left node.

- If this is unsuccessful, then join the 2 nodes and intervening parent key.
- If parent ends up with less than critical number of keys then repeat this on the parent.