Module 5

Graphs - Terminology and Representation

Definitions: Graph, Vertices, Edges

A graph G consists of two sets V and E. V is finite and non empty ,E is a set of pair of vertices and these pairs are also called as edges

Graph $G = (V, E)$ by defining a pair of sets:

1. V = a set of vertices
2. E = a set of edges

Vertices:

- Vertices also called nodes
- Denote vertices with labels

Edges:

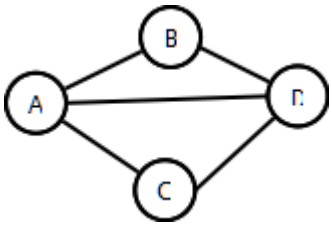
- Each edge is defined by a pair of vertices
- An edge connects the vertices that define it
- In some cases, the vertices can be the same

Representation:

- Represent vertices with circles, perhaps containing a label
- Represent edges with lines between circles

Example:

- V = {A,B,C,D}
- E = {(A,B),(A,C),(A,D),(B,D),(C,D)}



Examples:

- Cities with distances between
- Roads with distances between intersection points
- Course prerequisites
- Network
- Social networks
- Program call graph and variable dependency graph

Graph Classifications

- There are several common kinds of graphs
  - Weighted or unweighted
  - Directed or undirected
  - Cyclic or acyclic

Types of Graphs: Weighted and Unweighted

Graphs can be classified by whether or not their edges have weights

Weighted graph: edges have a weight

  - Weight typically shows cost of traversing
  - Example: weights are distances between cities
- Unweighted graph: edges have no weight
  - Edges simply show connections
  - Example: course prereqs

Types of Graphs: Directed and Undirected

- Graphs can be classified by whether or their edges are have direction

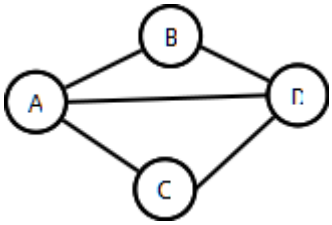
Undirected Graphs: each edge can be traversed in **either direction**

Directed Graphs: each edge can be traversed **only in a specified direction**

Undirected Graphs

- **Undirected Graph**: no implied direction on edge between nodes

  - The example from above is an undirected graph



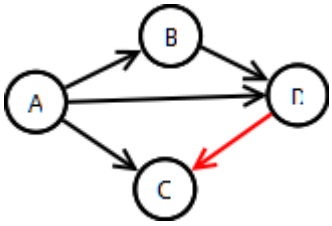
  - In diagrams, edges have no direction (ie they are not arrows)
  - Can traverse edges in either directions

In an undirected graph, an edge is an **unordered** pair

  - Actually, an edge is a set of 2 nodes, but for simplicity we write it with parens
    - For example, we write (A, B) instead of {A, B}
    - Thus, (A,B) = (B,A), etc
    - If (A,B) ∈ E then (B,A) ∈ E
  - Formally: ∀ u,v ∈ E, (u,v)=(v,u) and u ≠ v
- A node normally does not have an edge to itself

Directed Graphs

- **Digraph**: A graph whose edges are directed (ie have a direction)
  - Edge drawn as arrow
  - Edge can only be traversed in direction of arrow
  - Example: E = {(A,B), (A,C), (A,D), (B,C), (D,C)}



  - Examples: courses and prerequisites, program call graph
- In a digraph, an edge is an **ordered** pair
  - Thus: (u,v) and (v,u) are not the same edge
  - In the example, (D,C) ∈ E, (C,D) ∉ E

- What would edge (B,A) look like? Remember (A,B) ≠ (B,A)
- A node can have an edge to itself (eg (A,A) is valid)

Subgraph

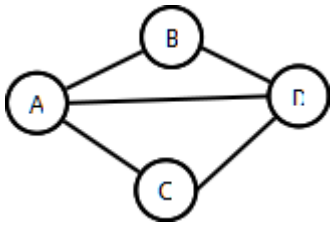
- If graph G=(V, E)
  - Then Graph G'=(V',E') is a **subgraph** of G if V' ⊆ V and E' ⊆ E and
- Example ...

Degree of a Node

- The **degree** of a node is the number of edges the node is used to define
- In the example above:
- Degree 2: B and C
- Degree 3: A and D
- A and D have **odd degree**, and B and C have **even degree**
- Can also define **in-degree** and **out-degree**
- In-degree: Number of edges pointing **to** a node
- Out-degree: Number of edges pointing **from** a node
- Whare are the in- and out-degree of the example?

Graphs: Terminology with Paths

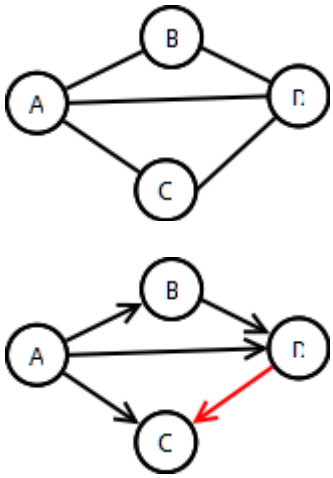
- **Path**: sequence of vertices in which each pair of successive vertices is connected by an edge
- **Cycle**: a path that starts and ends on the same vertex
- **Simple path**: a path that does not cross itself
- That is, no vertex is repeated (except first and last)
- Simple paths cannot contain cycles
- **Length** of a path: Number of edges in the path
- Sometimes the sum of the weights of the edges
- Examples



- A sequence of vertices: (A, B, C, D) [Is this path, simple path, cycle?]
- (A, B, D, A, C) [path, simple path, cycle?]
- (A, B, D, A, C) [path, simple path, cycle?]
- Cycle: ?
- Simple Cycle: ?
- Lengths?

Cyclic and Acyclic Graphs

- A **Cyclic** graph contains cycles
  - Example: roads (normally)
- An **acyclic** graph contains no cycles
  - Example: Course prereqs!
- Examples - Are these cyclic or acyclic?



Connected and Unconnected Graphs and Connected Components

- An *undirected* graph is **connected** if every pair of vertices has a path between it
  - Otherwise it is unconnected
  - Give an example of a connected graph
- An unconnected graph can be broken in to **connected components**
- A *directed* graph is **strongly connected** if every pair of vertices has a path between them, in **both directions**

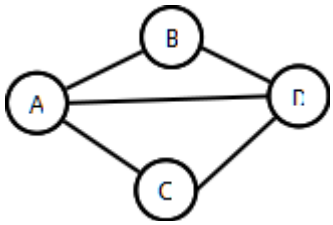
Data Structures for Representing Graphs

- Two common data structures for representing graphs:
  - Adjacency lists
  - Adjacency matrix

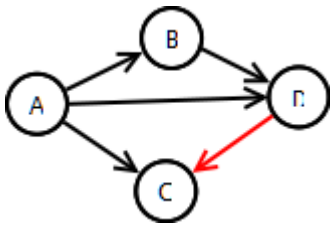
Adjacency List Representation

- Each node has a list of adjacent nodes
- Example (undirected graph):
  - A: B, C, D
  - B: A, D
  - C: A, D

- D: A, B, C



- Example (directed graph):
- A: B, C, D
- B: D
- C: Nil
- D: C



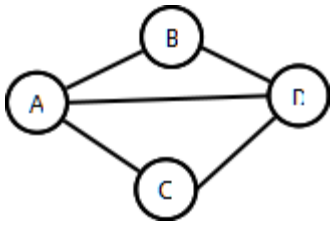
- Weighted graph can store weights in list
- Space: Θ(V + E) (ie |V| + |E|)

Adjacency Matrix Representation

- **Adjacency Matrix**: 2D array containing weights on edges
- Row for each vertex
- Column for each vertex
- Entries contain weight of edge from row vertex to column vertex
- Entries contain ∞ (ie Integer'last) if no edge from row vertex to column vertex
- Entries contain 0 on diagonal (if self edges not allowed)
- Example undirected graph (assume self-edges not allowed):

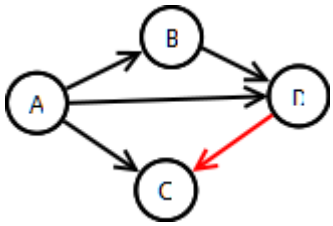
	A	B	C	D
A	0	1	1	1
B	1	0	999	1
C	1	999	0	1
D	1	1	1	0

-

- 

- Example directed graph (assume self-edges allowed):

	A	B	C	D
A	999	1	1	1
B	999	999	999	1
C	999	999	999	999
D	999	999	1	999



- Can store weights in cells

Insertion sort

```
#include<stdio.h>
int main(){

  int i,j,s,temp,a[20];

  printf("Enter total elements: ");
  scanf("%d",&s);

  printf("Enter %d elements: ",s);
  for(i=0;i<s;i++)
     scanf("%d",&a[i]);

  for(i=1;i<s;i++){
     temp=a[i];
     j=i-1;
     while((temp<a[j])&&(j>=0)){
     a[j+1]=a[j];
```

```
            j=j-1;
        }
        a[j+1]=temp;
    }

    printf("After sorting: ");
    for(i=0;i<s;i++)
        printf(" %d",a[i]);

    return 0;
}
```

Output:
Enter total elements: 5
Enter 5 elements: 3 7 9 0 2
After sorting: 0 2 3 7 9

Radix sort

```
#include <stdio.h>

#define MAX 100

#define SHOWPASS

void print(int *a, int n) {

    int i;

    for (i = 0; i < n; i++)

      printf("%d\t", a[i]);

}

void radix_sort(int *a, int n) {

    int i, b[MAX], m = 0, exp = 1;

    for (i = 0; i < n; i++) {

                if (a[i] > m)

                  m = a[i];
```

```
    }
    while (m / exp > 0) {
        int box[10] = {
            0
        }
        ;
        for (i = 0; i < n; i++)
          box[a[i] / exp % 10]++;
        for (i = 1; i < 10; i++)
          box[i] += box[i - 1];
        for (i = n - 1; i >= 0; i--)
          b[--box[a[i] / exp % 10]] = a[i];
        for (i = 0; i < n; i++)
          a[i] = b[i];
        exp *= 10;
        #ifdef SHOWPASS
          printf("\n\nPASS   : ");
        print(a, n);
        #endif
    }
}
int main() {
    int arr[MAX];
```

```
    int i, num;
    printf("\nEnter total elements (num < %d) : ", MAX);
    scanf("%d", &num);
    printf("\nEnter %d Elements : ", num);
    for (i = 0; i < num; i++)
      scanf("%d", &arr[i]);
    printf("\nARRAY  : ");
    print(&arr[0], num);
    radix_sort(&arr[0], num);
    printf("\n\nSORTED  : ");
    print(&arr[0], num);
    return 0;
}
```

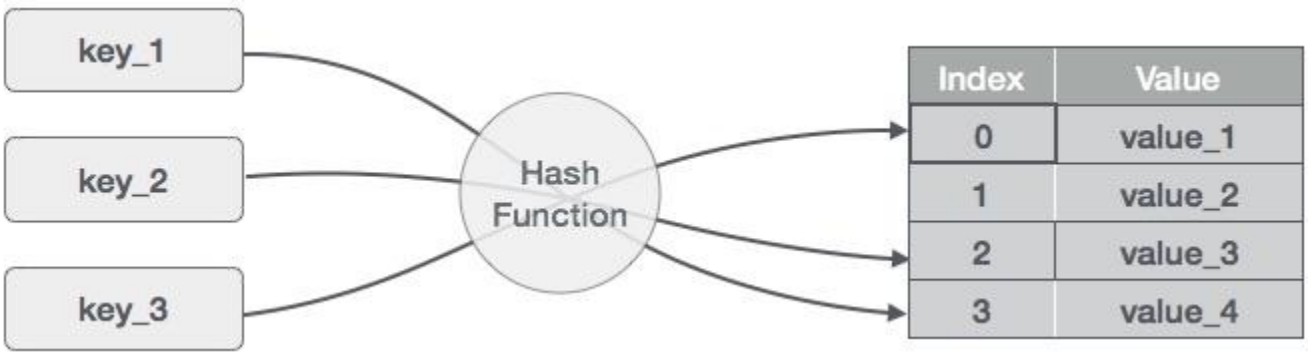
Hashing

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data values has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hashtable of size 20, and following items are to be stored. Item are in (key,value) format.



(1,20)

(2,70)

(42,80)

(4,25)

(12,44)

(14,32)

(17,11)

(13,78)

(37,98)

S.n.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique used create already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

S.n.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are basic primary operations of a hashtable which are following.

- **Search** − search an element in a hashtable.
- **Insert** − insert an element in a hashtable.
- **delete** − delete an element from a hashtable.

DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem {
   int data;
```

```
  int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
  return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched. Compute the hash code of the key passed and locate the element using that hashcode as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```
struct DataItem *search(int key){
  //get the hash
  int hashIndex = hashCode(key);

  //move in array until an empty
  while(hashArray[hashIndex] != NULL){

    if(hashArray[hashIndex]->key == key)
      return hashArray[hashIndex];

    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
  }

  return NULL;
}
```

Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

```
void insert(int key,int data){
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1){
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hashtable intact.

```
struct DataItem* delete(struct DataItem* item){
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL){

      if(hashArray[hashIndex]->key == key){
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
```

```
      hashArray[hashIndex] = dummyItem;
      return temp;
    }

    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
  }

  return NULL;
}
```

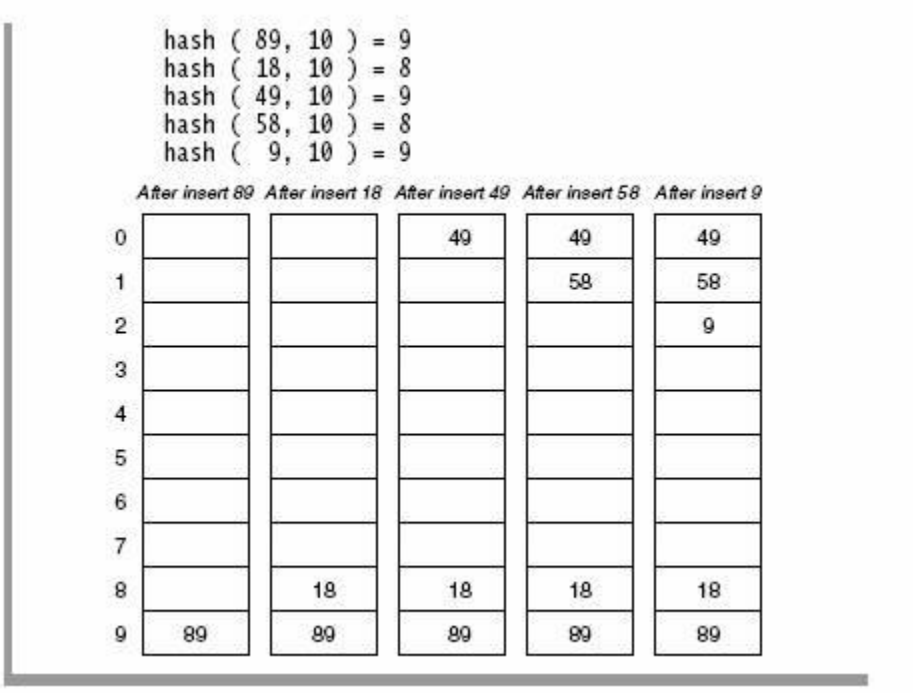
Collision Resolution

Introduction

In this lesson we will discuss several collision resolution strategies. The key thing in hashing is to find an easy to compute hash function. However, collisions cannot be avoided. Here we discuss three strategies of dealing with collisions, linear probing, quadratic probing and separate chaining.

Linear Probing

Suppose that a key hashes into a position that is already occupied. The simplest strategy is to look for the next available position to place the item. Suppose we have a set of hash codes consisting of {89, 18, 49, 58, 9} and we need to place them into a table of size 10. The following table demonstrates this process.



The first collision occurs when 49 hashes to the same location with index 9. Since 89 occupies the A[9], we need to place 49 to the next available position. Considering the array as circular, the next available position is 0. That is (9+1) mod 10. So we place 49 in A[0]. Several more collisions occur in this simple example and in each case we keep looking to find the next available location in the array to place the element. Now if we need to find the element, say for example, 49, we first compute the hash code (9), and look in A[9]. Since we do not find it there, we look in A[(9+1) % 10] = A[0], we find it there and we are done. So what if we are looking for 79? First we compute hashcode of 79 = 9. We probe in A[9], A[(9+1)%10]=A[0], A[(9+2)%10]=A[1], A[(9+3)%10]=A[2], A[(9+4)%10]=A[3] etc. Since A[3] = null, we do know that 79 could not exists in the set.