CMR INSTITUTE OFTECHNOLOGY

USN

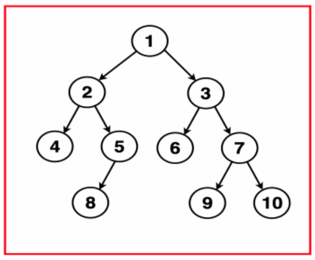
Internal Assessment Test III – May2024

Sub:	**Data Structures**							**Sub Code:**	**22MCA13**
Date:	**21/05/2024**	**Duration:**	**90 min's**	**Max Marks:**	**50**	**Sem:**	**I**	**Branch:**	**MCA**

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

		MARKS	OBE	
			CO	RBT
	PART I			
1	Write a C program to simulate the working of a singly linked list providing the following operations: a. Insert begin/ insert last b. Delete from the beginning/end d. Display.	[10]	CO1	L1
	OR			
2	Write a program to implement stack operations push(), pop() and Display() using singly linked list	[10]	CO1	L2
	PART II			
3	Define a binary tree. With example show array and linked representation of binary tree . Discuss the disadvantages of Array Representation	[10]	CO2	L2
	OR			
4	Mention different types of binary trees and explain them briefly. With example explain the following i) Degree of a node, ii)Level of a binary tree iii)Siblings.	[10]	CO2	L2

PART III

5.	Write the C-routines to traverse the tree using i) Inorder ii) Pre-order iii) Post-order. Also find the traversals for the given tree: 	[10]	CO2	L2
	OR			
6.	What is a graph? Write the terminologies used in graph. Explain adjacency matrix and adjacency list representation of graphs with example.	[10]	CO3	L2
	PART IV			
7.	i) Construct a binary search tree for inputs 22, 14, 18, 50, 9, 15, 7, 6, 12, 32, 25 ii) Construct a binary tree where Preorder and Inorder of a traversal yields the following sequence of nodes. Inorder: 8,4,10,9,11,2,5,1,6,3,7 Preorder:1,2,4,8,9,10,11,5,3,6,7	[10]	CO2	L2
	OR			
8.	What is threaded binary tree? Write the rules to construct the threads and explain with example.	[10]	CO2	L2
	PART V			
9.	Sort the numbers given below using radix sort and insertion sort 345, 654, 924, 123, 567, 472, 555, 808, 911 with appropriate figure. **OR**	[10]	CO3	L3
10.	What is collision? Explain various methods for resolving Hash collisions.	[10]	CO2	L3

1. Write a C program to simulate the working of a singly linked list providing the following operations: a. Insert begin/ insert last b. Delete from the beginning/end d. Display

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
 int data;
 struct node *next;
};

struct node *head;
void beginsert ();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();

void main ()
{
 int choice =0;
 while(choice != 9)
 {
  printf("\n\n*********Main Menu*********\n");

 printf("\nChoose one option from the following list ...\n");
 printf("\n===============================================\n");
 printf("\n1.Insert in begining\n
  2.Delete from Beginning\n
  3.Delete from last\n
  4.Delete node after specified location\n
  5.Search for an element\n
  6.Show\n7.Exit\n");

 printf("\nEnter your choice?\n");
 scanf("\n%d",&choice);
 switch(choice)
 {
  case 1:
   beginsert();
   break;
  case 2:
   begin_delete();
   break;
  case 3:
   last_delete();
   break;
  case 4:
   random_delete();
   break;
  case 5:
   search();
   break;
  case 6:
   display();
   break;
  case 7:
    exit(0);
    break;
   default:
    printf("Please enter valid choice..");
  }
 }
}
void beginsert()
{
 struct node *ptr;
 int item;
 ptr = (struct node *) malloc(sizeof(struct node *));



 if(ptr == NULL)
 {
  printf("\nOVERFLOW");
 }
 else
 {
  printf("\nEnter value\n");
  scanf("%d",&item);
  ptr->data = item;
  ptr->next = head;
  head = ptr;
  printf("\nNode inserted");
 }
}
void begin_delete()
{
 struct node *ptr;

if(head == NULL)
{
 printf("\nList is empty\n");
}
else
{
 ptr = head;
 head = ptr->next;
 free(ptr);

  printf("\nNode deleted from the begining ...\n");
 }
}
void last_delete()
{
 struct node *ptr,*ptr1;
 if(head == NULL)
 {
  printf("\nlist is empty");
 }
 else if(head -> next == NULL)
 {
  head = NULL;
  free(head);
  printf("\nOnly node of the list deleted ...\n");
 }
 else
 { ptr = head;
  while(ptr->next != NULL)
   {
    ptr1 = ptr;
    ptr = ptr ->next;
   }
  ptr1->next = NULL;
  free(ptr);
  printf("\nDeleted Node from the last ...\n");
 }
}
void random_delete()
{
 struct node *ptr,*ptr1;
 int loc,i;
 printf("\n Enter the location of the node after which you want to perform deletion \n");
 scanf("%d",&loc);
 ptr=head;
 for(i=0;i<loc;i++)
  {
   ptr1 = ptr;
   ptr = ptr->next;
   if(ptr == NULL)
    {
     printf("\nCan’t delete");
     return;
    }
  }
 ptr1 ->next = ptr ->next;
 free(ptr);
 printf("\nDeleted node %d ",loc+1);
}

void search()
{
 struct node *ptr;

int item,i=0,flag=1;
ptr = head;
if(ptr == NULL)
{
 printf("\nEmpty List\n");
}
else
{
 printf("\nEnter item which you want to search?\n");
 scanf("%d",&item);

 while (ptr!=NULL)
  {
```

```
  if(ptr->data == item)
  {
   printf("item found at location %d ",i+1);
   flag=0;
                   break;
  }

  i++;
  ptr = ptr -> next;
 }
 if(flag==1)
    printf("Item not found\n");

 }
}
void display()
{
 struct node *ptr;
 ptr = head;
 if(ptr == NULL)
 { printf("Nothing to print");
 }
 else
 { printf("\nprinting values . . . . .\n");
  while (ptr!=NULL)
  {
   printf("\n%d",ptr->data);
   ptr = ptr -> next;
  }
      }
}
```

2. Write a program to implement stack operations push(), pop() and Display() using singly linked list

```
#include <stdio.h>
#include <stdlib.h>
void push(); void
pop(); void
display(); struct
node
{ int
val;
struct node *next;
};
struct node *head;
void main ()
{
int choice=0;
printf("\n*********Stack operations using linked list*********\n"); printf("\n----------------
------------------------------\n");
while(choice != 4)
 {
printf("\n\nChose one from the below options...\n");
printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
printf("\n Enter your choice \n"); scanf("%d",&choice);
switch(choice)
 {
case 1:
 {
push(); break;
```

```
 } case
2:
 {
pop(); break;
 } case
3:
 {
display(); break;
 } case
4:
 {
printf("Exiting...."); break;
 } default:
 {
printf("Please Enter valid choice ");
 }
 };
} } void push () { int val; struct node *ptr = (struct
node*)malloc(sizeof(struct node));
if(ptr == NULL)
 {
printf("not able to push the element");
 }
else
 {
printf("Enter the value");
scanf("%d",&val); if(head==NULL)
 {
ptr->val = val; ptr -
> next = NULL;
head=ptr;
 }
else
 {
ptr->val = val; ptr->next
= head; head=ptr;
 }
printf("Item pushed");
 } }
void pop() { int
item; struct node
*ptr; if (head ==
NULL)
 {
printf("Underflow");
 }
else
 {
item = head->val; ptr
= head;
head = head->next;
free(ptr); printf("Item
popped");
 } } void
display() {
int i; struct
node *ptr;
ptr=head; if(ptr
== NULL)
 {
```

```
printf("Stack is empty\n");
 }
else
 {
printf("Printing Stack elements \n");
while(ptr!=NULL)
 {
printf("%d\n",ptr->val); ptr
= ptr->next;
 }
 } }
```

3. Define a binary tree. With example show array and linked representation of binary tree . Discuss the disadvantages of Array Representation

BINARY TREES

A binary tree is a special kind of tree which can be easily maintained in the computer. Although such a tree may seem to be very restrictive, more general trees may be viewed as binary trees.

A **binary tree** T is defined as a finite set of elements, called **nodes**, such that:

i) T is empty (called the **null tree** or **empty tree**), or
ii) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.

a) If T does contain a root R, then the two trees T_1 and T_2 are called, respectively, the **left and right subtrees** of R.
b) If T_1 is nonempty, then its root is called the **left successor** of R; similarly, if T_2 is nonempty, then its root is called the **right successor** of R.

Binary Tree Representations: Array and linked Representation of Binary Trees

Array Representation

The array or sequential representation of a tree uses a single one-dimensional array, TREE as follows:

1) The root R of T is stored in TREE[1].
2) If a node N occupies TREE[K], then its left child is stored in TREE[2*K] and its right child in TREE[2* K+1].

Location 0 is not used.

NULL is used to indicate an empty subtree. TREE[1] = NULL indicates the tree is empty.

In other words, the nodes are numbered from 1 to n as per a full binary tree. Location 0 is not used. The nodes are then stored in a one-dimensional array whose position 0 is left e mpty, and the node numbered x is mapped to position with index x in the array. We can easily determine the locations of the parent, left child and right child of any node, i, in the binary tree using the following rule:

If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$,

1) parent(i) is at $\lfloor i/2 \rfloor$ if $i \ne 1$. If $i = 1$, i is at the root and has no parent.
2) leftChild(i) is at 2i if $2i \le n$. If $2i > n$, then I has no left child.
3) rightChild(i) is at 2i+1 if $2i+1 \le n$. If $2i+1 > n$, then I has no right child.

The sequential representation of the binary tree T in Fig. 4.8(a) appears in Fig. 4.8(b).

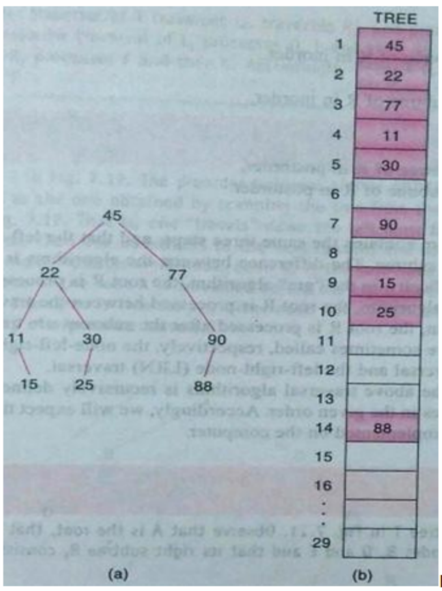


Fig. 4.8 Array Representation of Binary Tree

From the Fig.4.8 we see that even though T has only 9 nodes, 14 locations are required to represent the tree in the array.

4.

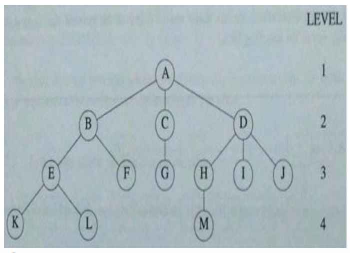
Mention different types of binary trees and explain them briefly. With example explain the following i) Degree of a node, ii) Level of a binary tree iii)Siblings.

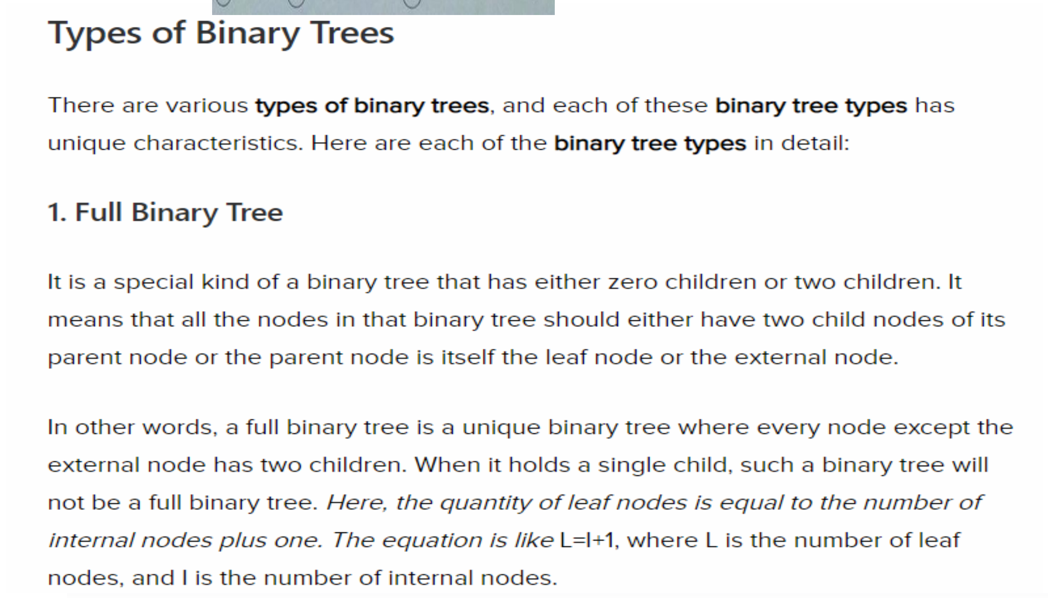
The tree is a nonlinear data structure. This structure is mainly used to represent data containing a hierarchical relationship between elements, e.g., records, family trees and tables of contents. A tree structure means that the data are organized in a hierarchical manner.

Definition: A tree is a finite set of one or more nodes such that

i) There is a specially designated node called the **root**.

ii) The remaining nodes are partitioned into n≥0 disjoint sets T_1, T_2, …, T_n, where each ofthese sets is a tree. T_1, T_2, …, T_n, are called the **subtrees** of the root.





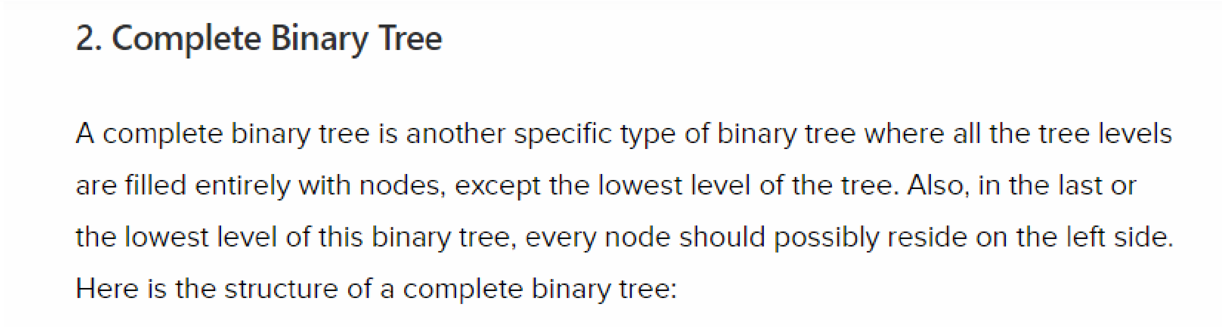
Types of Binary Trees

There are various **types of binary trees**, and each of these **binary tree types** has unique characteristics. Here are each of the **binary tree types** in detail:

1. Full Binary Tree

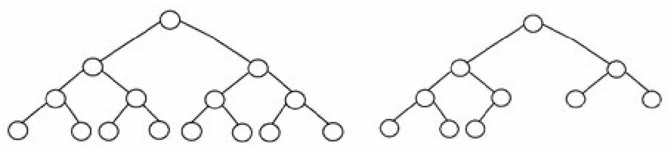
It is a special kind of a binary tree that has either zero children or two children. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.

In other words, a full binary tree is a unique binary tree where every node except the external node has two children. When it holds a single child, such a binary tree will not be a full binary tree. *Here, the quantity of leaf nodes is equal to the number of internal nodes plus one. The equation is like* L=I+1, where L is the number of leaf nodes, and I is the number of internal nodes.



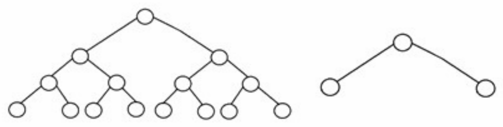
2. Complete Binary Tree

A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side. Here is the structure of a complete binary tree:



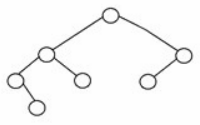
3. Perfect Binary Tree

A binary tree is said to be ‘perfect’ if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect binary tree having height ‘h’ has 2h – 1 node. Here is the structure of a perfect binary tree:



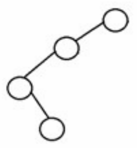
4. Balanced Binary Tree

A binary tree is said to be ‘balanced’ if the tree height is O(logN), where ‘N’ is the number of nodes. In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one. An AVL Tree and a Red-Black Tree are some common examples of data structure that can generate a balanced binary search tree. Here is an example of a balanced binary tree:



5. Degenerate Binary Tree

A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child. Such trees are similar to a linked list performance-wise. Here is an example of a degenerate binary tree:



There are many terms that are often used when referring to trees.

i) A **node** stands for the item of information plus the branches to other nodes.
Consider the tree in Fig.4.2. This tree has 13 nodes, each item of data being a single letter. The **root** is A, and we will normally draw trees with the root at the top.

ii) The number of subtrees of a node is called its **degree**.
The degree of A is 3, of C is I, and of F is zero.

iii) Nodes that have degree zero are called **leaf** or **terminal nodes**.
{K,L,F, G,M,I,J} is the set of leaf nodes.

iv) Consequently, the other nodes are referred to as **nonterminals**.

v) The roots of the subtrees of a node X are the **children** of X. X is the **parent** of its children.
Thus, the children of D are H, I, and J; the parent of D is A.

Every node N in a tree, except the root, has a unique parent, called the **predecessor** of N.

vi) Children of the same parent are said to be **siblings**.
H, I, and J are siblings.

vii) We can extend this terminology if we need to so that we can ask for the **grandparent** of M, which is D, and so on.

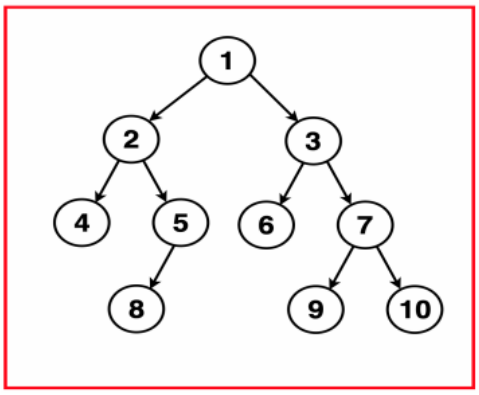
viii) A node L is called a **descendant** of a node N if there is a succession of children from N to L. N is called an **ancestor** of L.

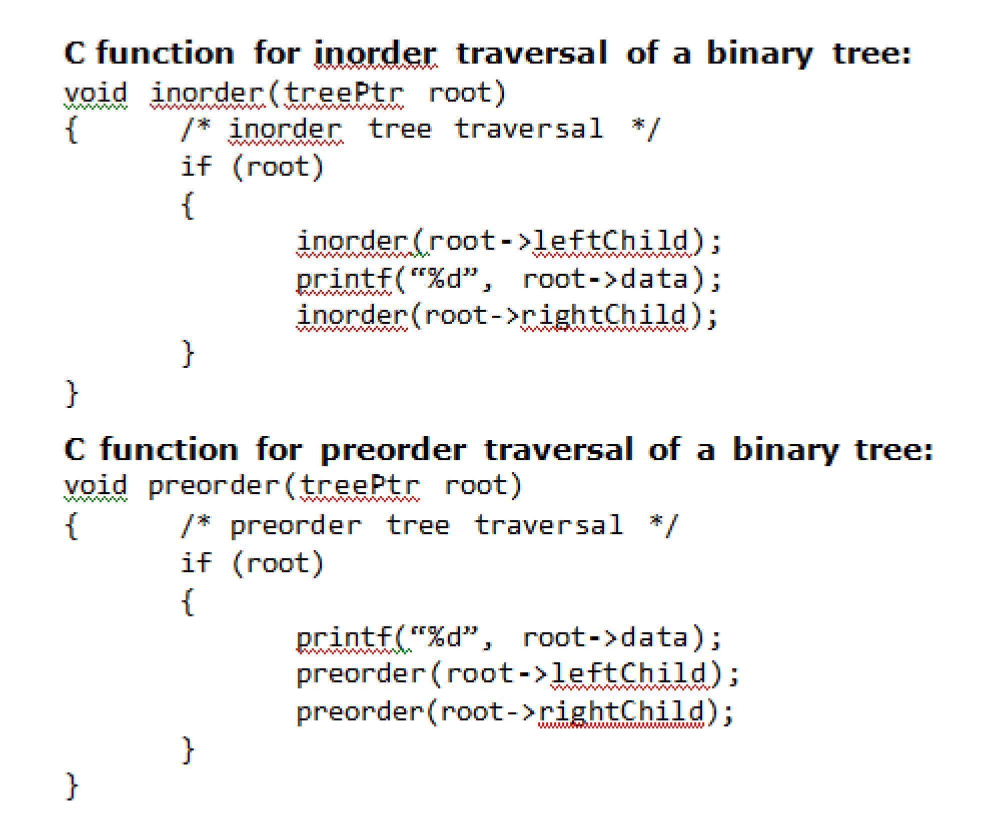
ix) The **degree of a tree** is the maximum of the degree of the nodes in the tree.
The tree of Fig.4.2 has degree 3. The ancestors of a node are all the nodes along the path from the root to that node. The ancestors of M are A, D, and H.

x) The **level of a node** is defined by letting the root be at level one. If a node is at level l, then its children are at level l+1.
Fig. 4.2 shows the level of all nodes in that tree.

xi) The **height** or **depth** of a tree is defined to be the maximum level of any node in the tree. Thus, the depth of the tree in Fig. 4.2 is 4.

5. Write the C-routines to traverse the tree using i) Inorder ii) Pre-order iii) Post-order. Also find the traversals for the given tree:





C function for inorder traversal of a binary tree:

```
void inorder(treePtr root)
{     /* inorder tree traversal */
      if (root)
      {
            inorder(root->leftChild);
            printf("%d", root->data);
            inorder(root->rightChild);
      }
}
```

C function for preorder traversal of a binary tree:

```
void preorder(treePtr root)
{     /* preorder tree traversal */
      if (root)
      {
            printf("%d", root->data);
            preorder(root->leftChild);
            preorder(root->rightChild);
      }
}
```

```
void postorder (struct node *root)
{
 if (root != NULL)
  {
   postorder (root->left);
   postorder (root->right);
   printf ("%d ", root->data);
  }
}
```

Solution
Preorder: 1 2 4 5 8 3 6 7 9 10
Inorder : 4 2 8 5 1 6 3 9 7 10
Post-order : 4 8 5 2 6 9 10 7 3 1

6. What is a graph? Write the terminologies used in graph.
Explain adjacency matrix and adjacency list representation of graphs with example
A graph is a mathematical structure used to model pairwise relations between objects. It consists of two main components:
Vertices (or Nodes): The fundamental units that represent objects in the graph.
Edges (or Links): The connections between pairs of vertices that represent the relationship between these objects.
Terminologies Used in Graph
Vertex (Node): A fundamental part of a graph, representing an entity.
Edge (Link): A connection between two vertices in a graph.
Adjacent Vertices: Two vertices that are connected by an edge.
Degree: The number of edges incident to a vertex. In directed graphs, we have:
In-degree: Number of incoming edges to a vertex.
Out-degree: Number of outgoing edges from a vertex.
Path: A sequence of edges that connects two vertices.
Cycle: A path that starts and ends at the same vertex without repeating any edge or vertex.
Connected Graph: A graph in which there is a path between every pair of vertices.
Subgraph: A graph formed from a subset of the vertices and edges of another graph.
Weighted Graph: A graph in which edges have weights or costs associated with them.
Directed Graph (Digraph): A graph where edges have a direction, going from one vertex to another.

Undirected Graph: A graph where edges have no direction.

Representations of Graph

Here are the two most common ways to represent a graph :

1. Adjacency Matrix
1. Adjacency List

Adjacency Matrix

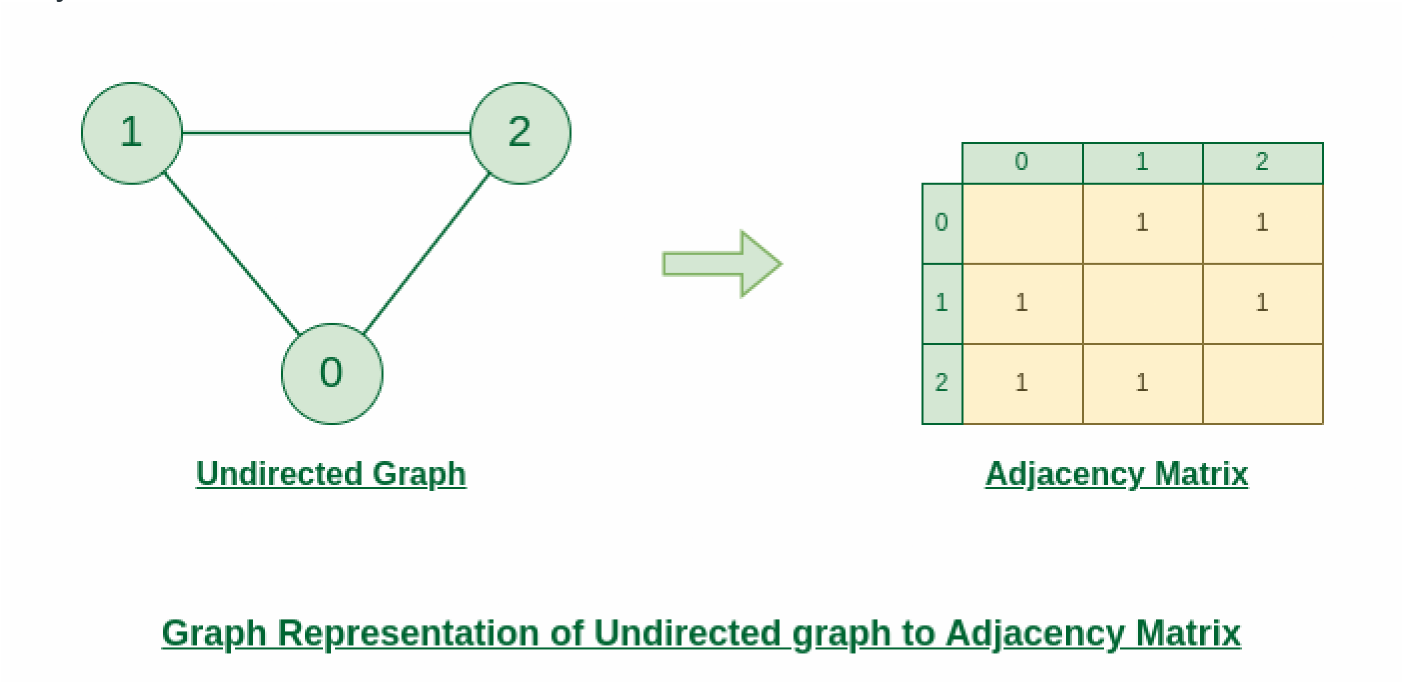
An adjacency matrix is a way of representing a graph as a matrix of boolean (0’s and 1’s).

Let’s assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- *If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.*
- *If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.*

Representation of Undirected Graph to Adjacency Matrix:

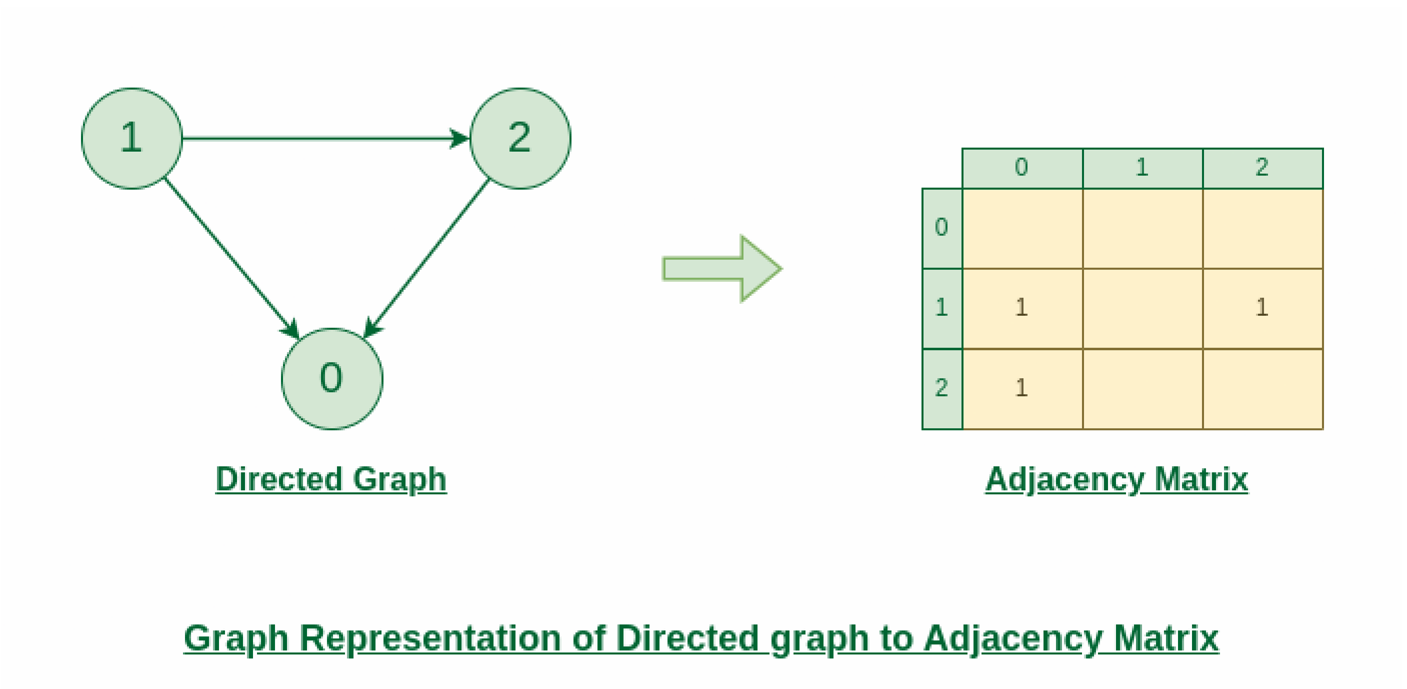
The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat[destination])** because we can go either way.



Undirected Graph to Adjacency Matrix

Representation of Directed Graph to Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.



Directed Graph to Adjacency Matrix

Adjacency List

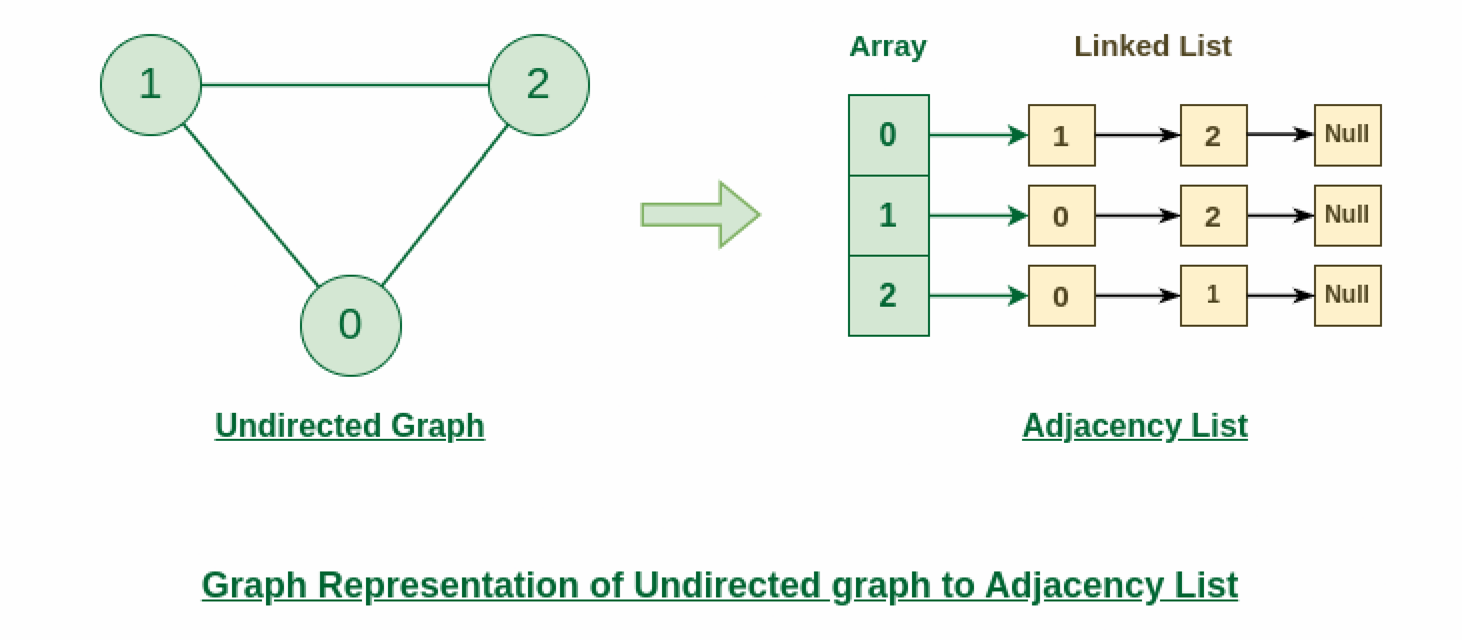
An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.

Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

- *adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.*
- *adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.*

Representation of Undirected Graph to Adjacency list:

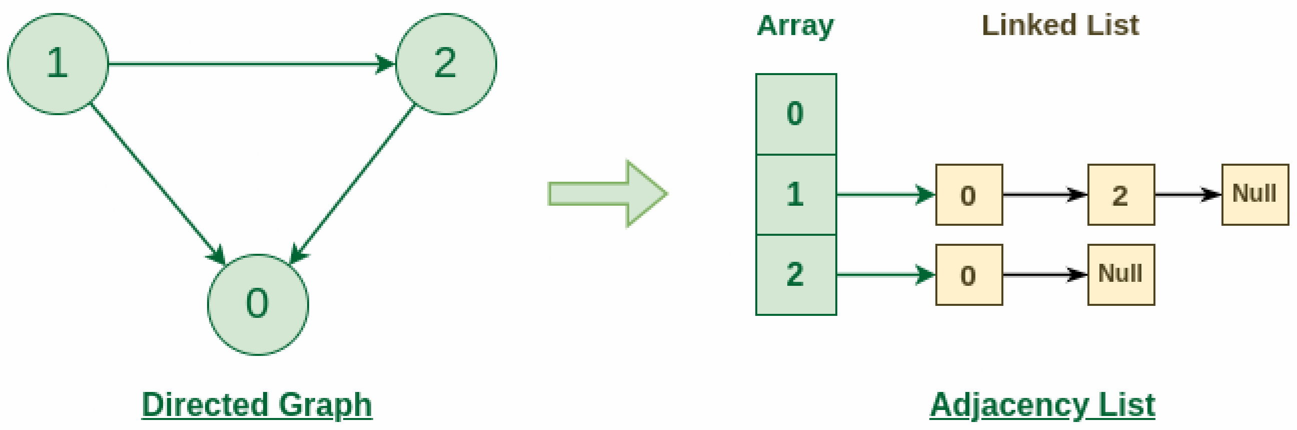
The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Undirected Graph to Adjacency list

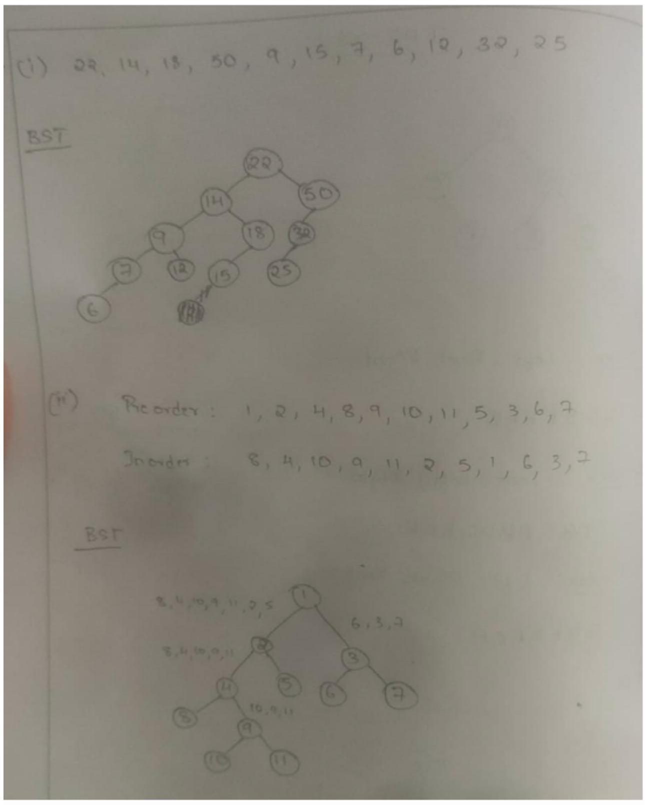
Representation of Directed Graph to Adjacency list:

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Directed graph to Adjacency List

7. **Construct a binary search tree for inputs 22, 14, 18, 50, 9, 15, 7, 6, 12, 32, 25**
   ii) Construct a binary tree where Preorder and Inorder of a traversal yields the following sequence of nodes. Inorder: 8,4,10,9,11,2,5,1,6,3,7 Preorder:1,2,4,8,9,10,11,5,3,6,7


(i) 22, 14, 18, 50, 9, 15, 7, 6, 12, 32, 25

BST

(ii) Preorder : 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7

Inorder : 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7

BST

8. **What is threaded binary tree? Write the rules to construct the threads and explain with example.**

Threaded binary trees

In a linked representation of any binary tree, there are more links than actual pointers i.e. there are n+1 null links out of 2n total links.

A.J.Perlis and C. Thornton devised threaded binary tree in which they replaced the null links by pointers called **threads** to other nodes in the tree.

To construct the threads, use the following rules:

i) If `ptr->leftChild` is null, replace `ptr->leftChild` with a pointer to the node that would be visited before `ptr` in an inorder traversal i.e. the null is replaced with a po inter to the *inorder predecessor* of `ptr`.

ii) If `ptr->rightChild` is null, replace `ptr->rightChild` with a pointer to the node that would be visited after `ptr` in an inorder traversal i.e. the null is replaced with a pointer to the *inorder successor* of `ptr`.

A **threaded binary tree** is a binary tree which contains threads i.e. addresses of some nodes which facilitate move ment in the tree.

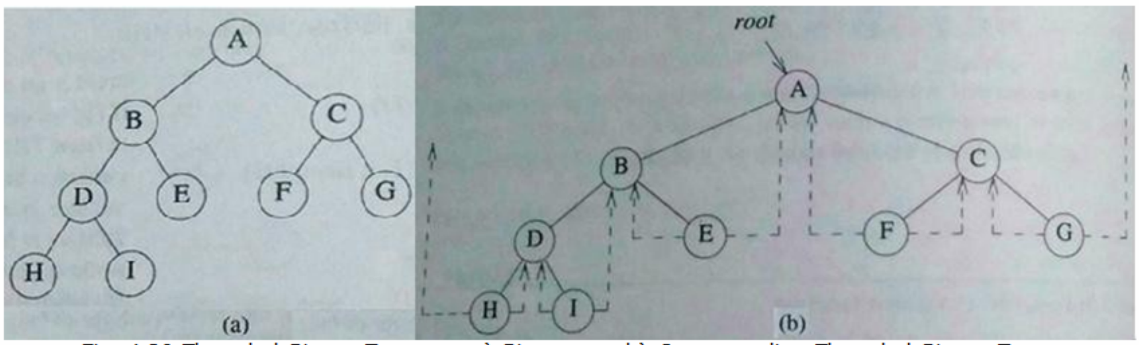


Fig. 4.20 Threaded Binary Tree a) Binary tree b) Corresponding Threaded Binary Tree

Fig. 4.20 shows a binary tree and the corresponding binary tree. The threads are shown as broken lines. This tree has 9 nodes and 10 threads. If we traverse the tree in inorder, the nodes will be visited in the order H, D, I, B, E, A, F, C, G. Example: node E has a predecessor thread that points to B and a successor thread that points to A.

When the tree is represented is memory, two additional fields, leftThread and rightThread are used to distinguish between threads and normal pointers. Assume that ptr is an arbitrary node in a threaded tree.

If ptr- >leftThread = TRUE, then ptr- >leftChild contains a thread; otherwise it contains a pointer to the left child.

Similarly, if ptr- >rightThread = TRUE, then ptr- >rightChild contains a thread; otherwise it contains a pointer to the right child.

This node structure is given by the following C declarations:

```
struct threadedTree
{
        short int leftThread;
        struct threadedTree *leftChild;
        int data;
        struct threadedTree *rightChild;
        short int rightThread;
 };
typedef struct threadedTree *threadedPTR;
```

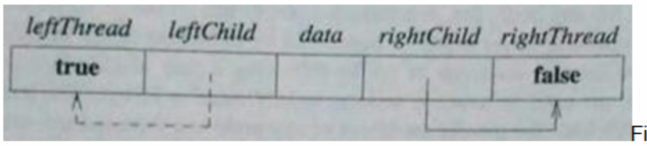


Fig. 4.21 An empty threaded binary tree

9. Sort the numbers given below using radix sort and insertion sort 345, 654, 924, 123, 567, 472, 555, 808, 911 with appropriate figure.

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it has the advantage of being simple to implement and efficient for small lists or arrays that are already mostly sorted.

Initial list: 345, 654, 924, 123, 567, 472, 555, 808, 911

Pass 1: (345 is already in the correct position)

List: 345, 654, 924, 123, 567, 472, 555, 808, 911

Pass 2: List: 345, 654, 924, 123, 567, 472, 555, 808, 911

Pass 3: List: 345, 654, 924, 123, 567, 472, 555, 808, 911

Pass 4: List: 123, 345, 654, 924, 567, 472, 555, 808, 911
Pass 5: List: 123, 345, 567, 654, 924, 472, 555, 808, 911
Pass 6: List: 123, 345, 472, 567, 654, 924, 555, 808, 911
Pass 7: List: 123, 345, 472, 555, 567, 654, 924, 808, 911
Pass 8: List: 123, 345, 472, 555, 567, 654, 808, 924, 911
Pass 9: List: 123, 345, 472, 555, 567, 654, 808, 911, 924
Final Sorted List:
123, 345, 472, 555, 567, 654, 808, 911, 924

Radix sort is a non-comparative sorting algorithm that sorts numbers by processing individual digits. It typically processes digits from the least significant to the most significant (LSD Radix Sort). Here's the step-by-step process of sorting the given list using radix sort:
Given list:
345, 654, 924, 123, 567, 472, 555, 808, 911
Radix Sort Process

1. **Identify the maximum number of digits:**
   - **The largest number is 924, which has 3 digits. Hence, we need to sort based on 3 digit positions (units, tens, and hundreds).**
2. **Sorting by the least significant digit (units place):**
   - **Given list: 345, 654, 924, 123, 567, 472, 555, 808, 911**
   - **Buckets based on units place:**
     0:
     1: 911
     2: 472
     3: 123
     4: 924
     5: 345, 555
     6: 654
     7: 567
     8: 808
     9:

   Combined list after sorting by units place:
   List: 911, 472, 123, 924, 345, 555, 654, 567, 808
3. **Sorting by the tens place:**
   - **Given list after previous step: 911, 472, 123, 924, 345, 555, 654, 567, 808**
   - **Buckets based on tens place:**
     0:
     1: 123
     2:
     3: 345
     4: 472
     5: 555
     6: 654, 567
     7:
     8: 808
     9: 911, 924

   Combined list after sorting by tens place:
   List: 123, 345, 555, 567, 654, 472, 808, 911, 924
4. **Sorting by the hundreds place:**
   - **Given list after previous step: 123, 345, 555, 567, 654, 472, 808, 911, 924**
   - **Buckets based on hundreds place:**
     0:
     1:
     2:
     3: 345
     4: 472
     5: 555, 567
     6: 654

7:
8: 808
9: 911, 924
Combined list after sorting by hundreds place:
List: 123, 345, 472, 555, 567, 654, 808, 911, 924
Final Sorted List:
123, 345, 472, 555, 567, 654, 808, 911, 924

10. What is collision? Explain various methods for resolving Hash collisions.

Hashing in data structure falls into a collision if two keys are assigned the same index number in the hash table. The collision creates a problem because each index in a hash table is supposed to store only one value. Hashing in data structure uses several collision resolution techniques to manage the performance of a hash table.
It is a process of finding an alternate location.
There are two types of collision resolution techniques.
- Separate chaining (open hashing)
- Open addressing (closed hashing)

Separate chaining: This method involves making a linked list out of the slot where the collision happened, then adding the new key to the list. Separate chaining is the term used to describe how this connected list of slots resembles a chain. It is more frequently utilized when we are unsure of the number of keys to add or remove.
Time complexity
- Its worst-case complexity for searching is o(n).
- Its worst-case complexity for deletion is o(n).

Advantages of separate chaining
- It is easy to implement.
- The hash table never fills full, so we can add more elements to the chain.
- It is less sensitive to the function of the hashing.

Disadvantages of separate chaining
- In this, the cache performance of chaining is not good.
- Memory wastage is too much in this method.
- It requires more space for element links.

Open addressing: To prevent collisions in the hashing table, open addressing is employed as a collision-resolution technique. No key is kept anywhere else besides the hash table. As a result, the hash table's size is never equal to or less than the number of keys. Additionally known as closed hashing.
The following techniques are used in open addressing:
- Linear probing
- Quadratic probing
- Double hashing

Linear probing: This involves doing a linear probe for the following slot when a collision occurs and continuing to do so until an empty slot is discovered.
The worst time to search for an element in linear probing is O. The cache performs best with linear probing, but clustering is a concern. This method's key benefit is that it is simple to calculate.
Disadvantages of linear probing:
- The main problem is clustering.
- It takes too much time to find an empty slot.

Quadratic probing: When a collision happens in this, we probe for the i2-nd slot in the i_{th} iteration, continuing to do so until an empty slot is discovered. In comparison to linear probing, quadratic probing has a worse cache performance. Additionally, clustering is less of a concern with quadratic probing.
Double hashing: In this, you employ a different hashing algorithm, and in the i_{th} iteration, you look for (i * hash 2(x)). The determination of two hash functions requires more time. Although there is no clustering issue, the performance of the cache is relatively poor when using double probing.