Binary Trees

Tree data structure a hierarchical data model in which data is linked together to form a representative tree.

Binary Trees are a type of tree where each node can have at most two children.

Binary Trees Example Code:

```
typedef struct TreeNode {
int data;
struct TreeNode* left;
struct TreeNode* right;
} TreeNode;
TreeNode* createNode(int x) {
TreeNode* t = (TreeNode*) malloc(sizeof(TreeNode));
t->data = x;
t->left = NULL;
t->right = NULL;
return t;
}
void printTree(TreeNode *root) {
if(root != NULL) {
printf("%d ", root->data);
printTree(root->left);
printTree(root->right);
}}
int main(void) {
TreeNode *root = createNode(5);
root->left = createNode(6);
root->right = createNode(8);
printTree(root);
getch();
```

Methods for navigating binary trees:

Preorder: Root-Left-Right

Inorder: Left-Root-Right

Postorder: Left-Right-Root

Methods for navigating Example Code:

```
void preorder(TreeNode *root) {
if(root != NULL) {
printf("%d ", root->data);
preorder(root->left);
preorder(root->right);
}}
void inorder(TreeNode *root) {
if(root != NULL) {
```

Binary Trees (cont)

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

Stack Trees

A stack tree is a data structure that allows to quickly find the smallest element in a data set. In this data structure, finding the smallest element, deleting the smallest element and adding elements to the tree can be done quickly.A binary tree that satisfies the following two properties is classified as a chunk tree data structure:

1. Tree integrity: All levels of the tree, except the last level, must be complete in terms of the nodes they contain. The nodes in the last level of the tree must also be full from left to right.

2. Heap property: The value of a node must be less than or equal to the values of its children.

Hash Tables

It is a data structure that stores data in the form of a key and data pair, allowing insertion, deletion and search operations to be performed very quickly. The general working logic is to store data in an N dimensional array and use a key consisting of a number or string to access the data. For a data to be stored, a key value is sent to the hash function, the value calculated by the function becomes the index where the data will be stored in the array.

Example of a hash function that takes a numeric value as the key:

```
int hash(int key, int N) {
return key % N;
}
```

Example of a hash function that takes a string value as the key:

```
int hash(char key[], int M, int N) {
int i, sum = 0;
for(i=0; i<M; i++) {
sum += key[i];
}
return sum % N;
}
```

Binary Search Trees

Binary search trees are a special type of binary tree. In this data structure, in addition to the binary tree properties, there is a size-m-inority relationship between the data in the nodes.
In order for a tree with binary tree properties to be a binary search tree, each node in the tree node must be greater than all values in its left subtree and less than or equal to all values in its right subtree.

Binary search tree structure and node insertion function example:

```
typedef struct TreeNode {
int data;
struct TreeNode *left;
struct TreeNode *right;
} TreeNode;
TreeNode* insertNode(TreeNode *node, int x) {
if(node == NULL) {
TreeNode *t = (TreeNode *) malloc(sizeof(TreeNode));
t -> data = x;
t -> left = t -> right = NULL;
return t;
}
else if(x > node->data) {
node->right = insertNode(node->right, x);
}
else {
node->left = insertNode(node->left, x);
}}
```

Example of node extraction function from binary search tree:

```
typedef struct TreeNode {
int data;
struct TreeNode *left;
struct TreeNode *right;
} TreeNode;
TreeNode* findMin(TreeNode *node) {
if(node == NULL)
return NULL;
if(node->left)
return findMin(node->left);
else
return node;
}
TreeNode* insertNode(TreeNode *node, int x) {
if(node == NULL) {
TreeNode *t = (TreeNode *) malloc(sizeof(TreeNode));
```

Binary Search Trees (cont)

```
t -> data = x;
t -> left = t -> right = NULL;
return t;
}
else if(x > node->data) {
node->right = insertNode(node->right, x);
}
else {
node->left = insertNode(node->left, x);
}}
TreeNode* deleteNode(TreeNode *node, int x) {
TreeNode *temp;
if(node == NULL) {
printf("No node found to delete.\n");
}
else if(x < node->data) {
node->left = deleteNode(node->left, x);
}
else if(x > node->data) {
node->right = deleteNode(node->right, x);
}
else {
if(node->right && node->left) {
temp = findMin(node->right);
node -> data = temp->data;
node -> right = deleteNode(node->right, temp->data);
}
else {
temp = node;
if(node->left == NULL)
node = node->right;
else if(node->right == NULL)
node = node->left;
free(temp);
}}
return node;
}
```