|  |  |
| --- | --- |
| **COURSE CODE:** DJS22ITL502 | **DATE:** 16-10-24 |
| **COURSE NAME:** Advanced Data Structures Laboratory | **CLASS:** TY B. TECH |
| **NAME:** Anish Sharma | **ROLL:** I011 |

**EXPERIMENT NO. 7**

**CO/LO:** Choose appropriate data structure and use it to design algorithm for solving a specific problem

**AIM / OBJECTIVE:** To implement various operations on a B-Tree.

**Properties of B-Tree:**

Balanced Tree: B-Trees remain balanced, ensuring that all leaf nodes are at the same level. M-Way Search Tree: Each node can have multiple keys and children (up to m), reducing tree height.

Efficient Disk Access: Designed for systems with slow disk access by minimizing the number of I/O operations.

Sorted Nodes: Keys within each node are stored in sorted order, making searching efficient. Variable Node Size: Nodes can grow or shrink as keys are inserted or deleted, within a defined minimum and maximum size.

**TECHNOLOGY STACK USED: C, C++, JAVA SOURCE CODE:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    #define MAX 3  #define MIN 2    struct BTreeNode { int val[MAX + 1], count;  struct BTreeNode\* link[MAX + 1]; };    struct BTreeNode\* root;    struct BTreeNode\* createNode(int val, struct BTreeNode\* child) { struct BTreeNode\* newNode = (struct BTreeNode\*)malloc(sizeof(struct  BTreeNode)); newNode->val[1] = val; newNode->count = 1; newNode->link[0] = root; newNode->link[1] = child; return newNode; |

|  |
| --- |
| }  void insertNode(int val, int pos, struct BTreeNode\* node, struct BTreeNode\* child) { int j = node->count; while (j > pos) { node->val[j + 1] = node->val[j]; node->link[j + 1] = node->link[j]; j--; } node->val[j + 1] = val; node->link[j + 1] = child; node->count++;  }  void splitNode(int val, int\* pval, int pos, struct BTreeNode\* node, struct  BTreeNode\* child, struct BTreeNode\*\* newNode) { int median, j; if (pos > MIN) median = MIN + 1;  else median = MIN;  \*newNode = (struct BTreeNode\*)malloc(sizeof(struct BTreeNode)); j = median + 1; while (j <= MAX) {  (\*newNode)->val[j - median] = node->val[j]; (\*newNode)->link[j - median] = node->link[j]; j++; } node->count = median;  (\*newNode)->count = MAX - median;  if (pos <= MIN) insertNode(val, pos, node, child);  else insertNode(val, pos - median, \*newNode, child);  \*pval = node->val[node->count];  (\*newNode)->link[0] = node->link[node->count]; node->count--;  }  int setValue(int val, int\* pval, struct BTreeNode\* node, struct BTreeNode\*\* child) { int pos; if (!node) {  \*pval = val;  \*child = NULL; |

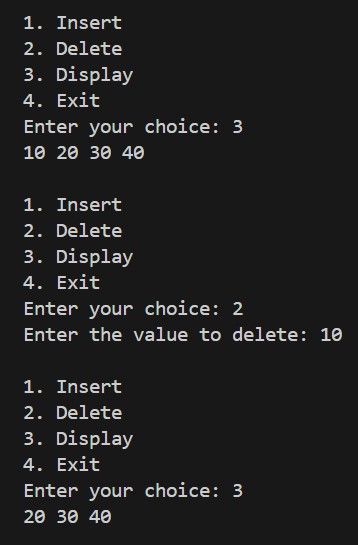
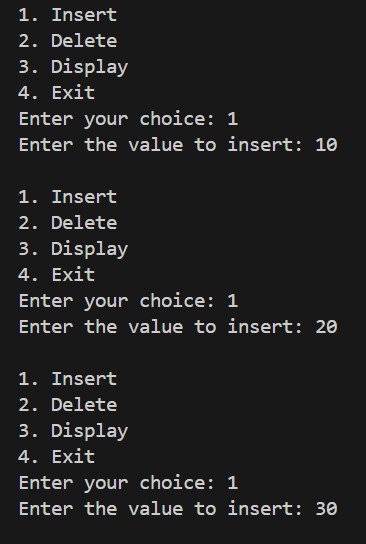
|  |
| --- |
| return 1;  }  if (val < node->val[1]) { pos = 0;  } else { for (pos = node->count; (val < node->val[pos] && pos > 1); pos--)  ; if (val == node->val[pos]) return 0;  }  if (setValue(val, pval, node->link[pos], child)) { if (node->count < MAX) { insertNode(\*pval, pos, node, \*child);  } else { splitNode(\*pval, pval, pos, node, \*child, child); return 1;  } }  return 0;  }  void insert(int val) { int flag, i; struct BTreeNode\* child;  flag = setValue(val, &i, root, &child); if (flag) root = createNode(i, child);  }  void copySuccessor(struct BTreeNode\* myNode, int pos) { struct BTreeNode\* dummy; dummy = myNode->link[pos];  while (dummy->link[0] != NULL) dummy = dummy->link[0];  myNode->val[pos] = dummy->val[1];  }  void removeVal(struct BTreeNode\* myNode, int pos) { int i = pos + 1; while (i <= myNode->count) { myNode->val[i - 1] = myNode->val[i]; myNode->link[i - 1] = myNode->link[i]; i++; } myNode->count--; |

|  |
| --- |
| }  void rightShift(struct BTreeNode\* myNode, int pos) { struct BTreeNode\* x = myNode->link[pos]; int j = x->count;  while (j > 0) { x->val[j + 1] = x->val[j]; x->link[j + 1] = x->link[j];  } x->val[1] = myNode->val[pos]; x->link[1] = x->link[0]; x->count++;  x = myNode->link[pos - 1]; myNode->val[pos] = x->val[x->count]; myNode->link[pos] = x->link[x->count]; x->count--;  }  void leftShift(struct BTreeNode\* myNode, int pos) { int j = 1; struct BTreeNode\* x = myNode->link[pos - 1];  x->count++; x->val[x->count] = myNode->val[pos]; x->link[x->count] = myNode->link[pos]->link[0];  x = myNode->link[pos]; myNode->val[pos] = x->val[1]; x->link[0] = x->link[1]; x- >count--;  while (j <= x->count) { x->val[j] = x->val[j + 1]; x->link[j] = x->link[j + 1]; j++;  } }  void mergeNodes(struct BTreeNode\* myNode, int pos) { int j = 1; struct BTreeNode\* x1 = myNode->link[pos], \*x2 = myNode->link[pos - 1];  x2->count++; x2->val[x2->count] = myNode->val[pos]; x2->link[x2->count] = x1->link[0]; while (j <= x1->count) { |

|  |
| --- |
| x2->count++; x2->val[x2->count] = x1->val[j]; x2->link[x2->count] = x1->link[j]; j++; }  for (j = pos; j < myNode->count; j++) { myNode->val[j] = myNode->val[j + 1]; myNode->link[j] = myNode->link[j + 1];  } myNode->count--; free(x1);  }  void adjustNode(struct BTreeNode\* myNode, int pos) { if (!pos) { if (myNode->link[1]->count > MIN) leftShift(myNode, 1);  else mergeNodes(myNode, 1);  } else { if (myNode->count != pos) { if (myNode->link[pos - 1]->count > MIN) rightShift(myNode, pos);  else if (myNode->link[pos + 1]->count > MIN) leftShift(myNode, pos + 1);  else mergeNodes(myNode, pos);  } else { if (myNode->link[pos - 1]->count > MIN) rightShift(myNode, pos);  else mergeNodes(myNode, pos); }  } }  int delValFromNode(int val, struct BTreeNode\* myNode) { int pos, flag = 0; if (myNode) { if (val < myNode->val[1]) { pos = 0; flag = 0;  } else { for (pos = myNode->count; (val < myNode->val[pos] && pos > 1); pos-  -)  ; if (val == myNode->val[pos]) flag = 1; |

|  |
| --- |
| else flag = 0;  }  if (flag) { if (myNode->link[pos - 1]) { copySuccessor(myNode, pos); flag = delValFromNode(myNode->val[pos], myNode->link[pos]); if (!flag) printf("Value not present in B-Tree\n");  } else removeVal(myNode, pos);  } else { flag = delValFromNode(val, myNode->link[pos]); if (myNode->link[pos]) { if (myNode->link[pos]->count < MIN) adjustNode(myNode, pos); }  } } return flag;  }  void delete(int val) { struct BTreeNode\* temp; if (!delValFromNode(val, root)) printf("Value %d is not found\n", val);  else { if (root->count == 0) { temp = root; if (root->link[0]) root = root->link[0];  else root = NULL;  free(temp);  }  } }  void inorder(struct BTreeNode\* myNode) { int i; if (myNode) { for (i = 0; i < myNode->count; i++) { inorder(myNode->link[i]);  printf("%d ", myNode->val[i + 1]);  } inorder(myNode->link[i]);  }  } |
| int main() { int val, choice; while (1) { printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice:  "); scanf("%d", &choice); switch (choice) { case 1:  printf("Enter the value to insert: "); scanf("%d", &val); insert(val); break;  case 2:  printf("Enter the value to delete: "); scanf("%d", &val); delete(val); break;  case 3:  inorder(root); printf("\n"); break;  case 4: exit(0);  default:  printf("Invalid choice\n"); }  }  } |

**OUTPUT:**



**CONCLUSION:** In this experiment we implemented various operations on a B-Tree

**REFERENCES:**

1. Peter Brass, “Advanced Data Structures”, Cambridge University Press, 2008

1. Robert Sedgewick & Kevin Wayne, “Algorithms”, 4th

Edition, Addison-Wesley Professional, 2011.