Binary Search Tree

```
#include <iostream>
#include<malloc.h>
using namespace std;
struct node {
  int key;
  struct node *left, *right;
};
struct node *newNode(int item) {
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}
void inorder(struct node *root) {
  if (root != NULL) {
    inorder(root->left);
    cout << root->key << " -> ";
    inorder(root->right);
  }
}
struct node *insert(struct node *node, int key) {
  if (node == NULL) return newNode(key);
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);
  return node;
}
```

```
struct node *minValueNode(struct node *node) {
  struct node *current = node;
  while (current && current->left != NULL)
    current = current->left;
    return current;
}
struct node *deleteNode(struct node *root, int key) {
  if (root == NULL) return root;
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    if (root->left == NULL) {
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }
    struct node *temp = minValueNode(root->right);
    root->key = temp->key;
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}
int main() {
  struct node *root = NULL;
```

```
  root = insert(root, 8);
  root = insert(root, 3);
  root = insert(root, 1);
  root = insert(root, 6);
  root = insert(root, 7);
  root = insert(root, 10);
  root = insert(root, 14);
  root = insert(root, 4);
  cout << "Inorder traversal: ";
  inorder(root);
  cout << "\nAfter deleting 10\n";
  root = deleteNode(root, 10);
  cout << "Inorder traversal: ";
  inorder(root);
}
```

OUTPUT:

```
Inorder traversal: 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 80 ->
After deleting 60
Inorder traversal: 20 -> 30 -> 40 -> 50 -> 70 -> 80 ->
--------------------------------
Process exited after 0.09862 seconds with return value 0
Press any key to continue . . .
```

Red black tree implementation

```
#include <iostream>
using namespace std;
struct Node {
  int data;
  Node *parent;
  Node *left;
  Node *right;
  int color;
};
typedef Node *NodePtr;
class RedBlackTree {
   private:
  NodePtr root;
  NodePtr TNULL;
  void initializeNULLNode(NodePtr node, NodePtr parent) {
    node->data = 0;
    node->parent = parent;
    node->left = nullptr;
    node->right = nullptr;
    node->color = 0;
  }
  void preOrderHelper(NodePtr node) {
    if (node != TNULL) {
      cout << node->data << " ";
      preOrderHelper(node->left);
      preOrderHelper(node->right);
    }
  }
```

```
void inOrderHelper(NodePtr node) {
  if (node != TNULL) {
    inOrderHelper(node->left);
    cout << node->data << " ";
    inOrderHelper(node->right);
  }
}
void postOrderHelper(NodePtr node) {
  if (node != TNULL) {
    postOrderHelper(node->left);
    postOrderHelper(node->right);
    cout << node->data << " ";
  }
}
NodePtr searchTreeHelper(NodePtr node, int key) {
  if (node == TNULL || key == node->data) {
    return node;
  }
  if (key < node->data) {
    return searchTreeHelper(node->left, key);
  }
  return searchTreeHelper(node->right, key);
}
void deleteFix(NodePtr x) {
  NodePtr s;
  while (x != root && x->color == 0) {
    if (x == x->parent->left) {
      s = x->parent->right;
      if (s->color == 1) {
        s->color = 0;
```

```
      x->parent->color = 1;
      leftRotate(x->parent);
      s = x->parent->right;
    }

    if (s->left->color == 0 && s->right->color == 0) {
      s->color = 1;
      x = x->parent;
    } else {
      if (s->right->color == 0) {
        s->left->color = 0;
        s->color = 1;
        rightRotate(s);
        s = x->parent->right;
      }
      s->color = x->parent->color;
      x->parent->color = 0;
      s->right->color = 0;
      leftRotate(x->parent);
      x = root;
    }
  } else {
    s = x->parent->left;
    if (s->color == 1) {
      s->color = 0;
      x->parent->color = 1;
      rightRotate(x->parent);
      s = x->parent->left;
    }
    if (s->right->color == 0 && s->right->color == 0) {
```

```
        s->color = 1;
        x = x->parent;
      } else {
        if (s->left->color == 0) {
          s->right->color = 0;
          s->color = 1;
          leftRotate(s);
          s = x->parent->left;
        }
        s->color = x->parent->color;
        x->parent->color = 0;
        s->left->color = 0;
        rightRotate(x->parent);
        x = root;
      }
    }
  }
  x->color = 0;
}
void rbTransplant(NodePtr u, NodePtr v) {
  if (u->parent == NULL) {
    root = v;
  } else if (u == u->parent->left) {
    u->parent->left = v;
  } else {
    u->parent->right = v;
  }
  v->parent = u->parent;
}
void deleteNodeHelper(NodePtr node, int key) {
```

```
NodePtr z = TNULL;
NodePtr x, y;
while (node != TNULL) {
  if (node->data == key) {
    z = node;
  }

  if (node->data <= key) {
    node = node->right;
  } else {
    node = node->left;
  }
}
if (z == TNULL) {
  cout << "Key not found in the tree" << endl;
  return;
}
y = z;
int y_original_color = y->color;
if (z->left == TNULL) {
  x = z->right;
  rbTransplant(z, z->right);
} else if (z->right == TNULL) {
  x = z->left;
  rbTransplant(z, z->left);
} else {
  y = minimum(z->right);
  y_original_color = y->color;
  x = y->right;
  if (y->parent == z) {
```

```
      x->parent = y;
    } else {
      rbTransplant(y, y->right);
      y->right = z->right;
      y->right->parent = y;
    }

    rbTransplant(z, y);
    y->left = z->left;
    y->left->parent = y;
    y->color = z->color;
  }
  delete z;
  if (y_original_color == 0) {
    deleteFix(x);
  }
}
void insertFix(NodePtr k) {
  NodePtr u;
  while (k->parent->color == 1) {
    if (k->parent == k->parent->parent->right) {
      u = k->parent->parent->left;
      if (u->color == 1) {
        u->color = 0;
        k->parent->color = 0;
        k->parent->parent->color = 1;
        k = k->parent->parent;
      } else {
        if (k == k->parent->left) {
          k = k->parent;
```

```
        rightRotate(k);
      }
      k->parent->color = 0;
      k->parent->parent->color = 1;
      leftRotate(k->parent->parent);
    }
  } else {
    u = k->parent->parent->right;

    if (u->color == 1) {
      u->color = 0;
      k->parent->color = 0;
      k->parent->parent->color = 1;
      k = k->parent->parent;
    } else {
      if (k == k->parent->right) {
        k = k->parent;
        leftRotate(k);
      }
      k->parent->color = 0;
      k->parent->parent->color = 1;
      rightRotate(k->parent->parent);
    }
  }
  if (k == root) {
    break;
  }
}
root->color = 0;
}
```

```
void printHelper(NodePtr root, string indent, bool last) {
  if (root != TNULL) {
    cout << indent;
    if (last) {
      cout << "R----";
      indent += "   ";
    } else {
      cout << "L----";
      indent += "|  ";
    }
    string sColor = root->color ? "RED" : "BLACK";
    cout << root->data << "(" << sColor << ")" << endl;
    printHelper(root->left, indent, false);
    printHelper(root->right, indent, true);
  }
}
 public:
RedBlackTree() {
  TNULL = new Node;
  TNULL->color = 0;
  TNULL->left = NULL;
  TNULL->right = NULL;
  root = TNULL;
}
void preorder() {
  preOrderHelper(this->root);
}
void inorder() {
  inOrderHelper(this->root);
}
```

```
void postorder() {
  postOrderHelper(this->root);
}
NodePtr searchTree(int k) {
  return searchTreeHelper(this->root, k);
}
NodePtr minimum(NodePtr node) {
  while (node->left != TNULL) {
    node = node->left;
  }
 return node;
}
NodePtr maximum(NodePtr node) {
  while (node->right != TNULL) {
    node = node->right;
  }
  return node;
}
NodePtr successor(NodePtr x) {
  if (x->right != TNULL) {
    return minimum(x->right);
  }
  NodePtr y = x->parent;
  while (y != TNULL && x == y->right) {
    x = y;
    y = y->parent;
  }
  return y;
}
NodePtr predecessor(NodePtr x) {
```

```
  if (x->left != TNULL) {
    return maximum(x->left);
  }
  NodePtr y = x->parent;
  while (y != TNULL && x == y->left) {
    x = y;
    y = y->parent;
  }
  return y;
}

void leftRotate(NodePtr x) {
  NodePtr y = x->right;
  x->right = y->left;
  if (y->left != TNULL) {
    y->left->parent = x;
  }
  y->parent = x->parent;
  if (x->parent ==NULL) {
    this->root = y;
  } else if (x == x->parent->left) {
    x->parent->left = y;
  } else {
    x->parent->right = y;
  }
  y->left = x;
  x->parent = y;
void rightRotate(NodePtr x) {
  NodePtr y = x->left;
  x->left = y->right;
```

```
  if (y->right != TNULL) {
    y->right->parent = x;
  }
  y->parent = x->parent;
  if (x->parent ==NULL) {
    this->root = y;
  } else if (x == x->parent->right) {
    x->parent->right = y;
  } else {
    x->parent->left = y;
  }
  y->right = x;
  x->parent = y;
}
void insert(int key) {
  NodePtr node = new Node;
  node->parent = NULL;
  node->data = key;
  node->left = TNULL;
  node->right = TNULL;
  node->color = 1;
  NodePtr y = NULL;
  NodePtr x = this->root;
  while (x != TNULL) {
    y = x;
    if (node->data < x->data) {
      x = x->left;
    } else {
      x = x->right;
    }
```

```
    }
    node->parent = y;
    if (y == NULL) {
      root = node;
    } else if (node->data < y->data) {
      y->left = node;
    } else {
      y->right = node;

    if (node->parent == NULL) {
      node->color = 0;
      return;
    }
    if (node->parent->parent == NULL) {
      return;
    }
    insertFix(node);
  }
  NodePtr getRoot() {
    return this->root;
  }
  void deleteNode(int data) {
    deleteNodeHelper(this->root, data);
  }
  void printTree() {
    if (root) {
      printHelper(this->root, "", true);
    }
  }
};
```

```
int main() {
  RedBlackTree bst;
  bst.insert(55);
  bst.insert(40);
  bst.insert(65);
  bst.insert(60);
  bst.insert(75);
  bst.insert(57);
  bst.printTree();
  cout << endl<< "After deleting" << endl;
  bst.deleteNode(40);
  bst.printTree();
}
```

OUTPUT:

```
R----55(BLACK)
   L----40(BLACK)
   R----65(RED)
      L----60(BLACK)
      |  L----57(RED)
      R----75(BLACK)

After deleting
R----65(BLACK)
   L----57(RED)
   |  L----55(BLACK)
   |  R----60(BLACK)
   R----75(BLACK)

--------------------------------
Process exited after 0.7436 seconds with return value 0
Press any key to continue . . .
```

Heap implementation

```
#include<iostream>
#include<vector>
using namespace std;
void swap(int*a,int*b)
{
	int temp=*b;
	*b=*a;
	*a=temp;
}
void heapify(vector<int>&hT,int i)
{
	int size=hT.size();
	int largest=i;
	int l=2*i+1;
	int r=2*i+2;
	if(l<size&&hT[l]>hT[largest])
	largest=l;
	if(r<size&&hT[r]>hT[largest])
	largest=r;
	if(largest!=i)
	{
		swap(&hT[i],&hT[largest]);
		heapify(hT,largest);
	}
}
void insert(vector<int>&hT,int newNum)
{
	int size=hT.size();
```

```
	if(size==0)
	{
		hT.push_back(newNum);
	}
	else
	{
		hT.push_back(newNum);
		for(int i=size/2-1;i>=0;i--)
		{
			heapify(hT,i);
		}
	}
}
void deleteNode(vector<int>&hT,int num)
{
	int size=hT.size();
	int i;
	for(i=0;i<size;i++)
	{
		if(num==hT[i])
		break;
	}
	swap(&hT[i],&hT[size-1]);
	hT.pop_back();
	for(int i=size/2-1;i>=0;i--)
	{
		heapify(hT,i);
	}
}
void printArray(vector<int>&hT)
```

```
{
	for(int i=0;i<hT.size();++i)
	cout<<hT[i]<<" ";
	cout<<"\n";
}
int main()
{
	vector<int>heapTree;
	insert(heapTree,3);
	insert(heapTree,4);
	insert(heapTree,9);
	insert(heapTree,5);
	insert(heapTree,2);
   cout<<"Max_heap array=";
   printArray(heapTree);
   deleteNode(heapTree,4);
   cout<<"After deleting an element:";
   printArray(heapTree);
}
```

OUTPUT:

```
Max_heap array=9 5 3 4 2
After deleting an element:9 5 3 2

--------------------------------
Process exited after 0.2062 seconds with return value 0
Press any key to continue . . .
```

Fibonacci heap

```
#include <cmath>
#include <cstdlib>
#include <iostream>
using namespace std;
struct node {
int n;
int degree;
node *parent;
node *child;
node *left;
node *right;
char mark;

char C;
};
class FibonacciHeap {
private:
int nH;
node *H;
public:
node *InitializeHeap();
int Fibonnaci_link(node *, node *, node *);
node *Create_node(int);
node *Insert(node *, node *);
node *Union(node *, node *);
node *Extract_Min(node *);
int Consolidate(node *);
int Display(node *);
```

```
node *Find(node *, int);
int Decrease_key(node *, int, int);
int Delete_key(node *, int);
int Cut(node *, node *, node *);
int Cascase_cut(node *, node *);
FibonacciHeap() { H = InitializeHeap(); }
};
node *FibonacciHeap::InitializeHeap() {
node *np;
np = NULL;
return np;
}
node *FibonacciHeap::Create_node(int value) {
node *x = new node;
x->n = value;
return x;
}
node *FibonacciHeap::Insert(node *H, node *x) {
x->degree = 0;
x->parent = NULL;
x->child = NULL;
x->left = x;
x->right = x;
x->mark = 'F';
x->C = 'N';
if (H != NULL) {
(H->left)->right = x;
x->right = H;
x->left = H->left;
H->left = x;
```

```
if (x->n < H->n)
H = x;
} else {
H = x;
}
nH = nH + 1;
return H;
}
int FibonacciHeap::Fibonnaci_link(node *H1, node *y, node *z) {
(y->left)->right = y->right;
(y->right)->left = y->left;
if (z->right == z)
H1 = z;
y->left = y;
y->right = y;
y->parent = z;
if (z->child == NULL)
z->child = y;
y->right = z->child;
y->left = (z->child)->left;
((z->child)->left)->right = y;
(z->child)->left = y;
if (y->n < (z->child)->n)
z->child = y;
z->degree++;
}
node *FibonacciHeap::Union(node *H1, node *H2) {
node *np;
node *H = InitializeHeap();
H = H1;
```

```
(H->left)->right = H2;
(H2->left)->right = H;
np = H->left;
H->left = H2->left;
H2->left = np;
return H;
}
int FibonacciHeap::Display(node *H) {
node *p = H;
if (p == NULL) {
cout << "Empty Heap" << endl;
return 0;
}
cout << "Root Nodes: " << endl;

do {
cout << p->n;
p = p->right;
if (p != H) {
cout << "-->";
}
} while (p != H && p->right != NULL);
cout << endl;
}
node *FibonacciHeap::Extract_Min(node *H1) {
node *p;
node *ptr;
node *z = H1;
p = z;
ptr = z;
```

```
if (z == NULL)
return z;
node *x;
node *np;
x = NULL;
if (z->child != NULL)
x = z->child;
if (x != NULL) {
x  ptr = x;
do {
np = x->right;
(H1->left)->right = x;
x->right = H1;
x->left = H1->left;
H1->left = x;
if (x->n < H1->n)
H1 = x;

x->parent = NULL;
x = np;
} while (np != ptr);
}
(z->left)->right = z->right;
(z->right)->left = z->left;
H1 = z->right;
if (z == z->right && z->child == NULL)
H = NULL;
else {
H1 = z->right;
Consolidate(H1);
```

```
}
nH = nH - 1;
return p;
}
int FibonacciHeap::Consolidate(node *H1) {
int d, i;
float f = (log(nH)) / (log(2));
int D = f;
node *A[D];
for (i = 0; i <= D; i++)
A[i] = NULL;
node *x = H1;
node *y;
node *np;
node *pt = x;
do {
pt = pt->right;
d = x->degree;
while (A[d] != NULL)
{
y = A[d];
if (x->n > y->n)
{
np = x;
x = y;
y = np;
}
if (y == H1)
H1 = x;
Fibonnaci_link(H1, y, x);
```

```
if (x->right == x)
H1 = x;
A[d] = NULL;
d = d + 1;
}
A[d] = x;
x = x->right;
}
while (x != H1);
H = NULL;
for (int j = 0; j <= D; j++) {
if (A[j] != NULL) {
A[j]->left = A[j];
A[j]->right = A[j];
if (H != NULL) {
(H->left)->right = A[j];
A[j]->right = H;
A[j]->left = H->left;
H->left = A[j];
if (A[j]->n < H->n)
H = A[j];
} else {
H = A[j];
}
if (H == NULL)
H = A[j];
else if (A[j]->n < H->n)
H = A[j];
}
}
```

```
}
int FibonacciHeap::Decrease_key(node *H1, int x, int k) {
node *y;
if (H1 == NULL) {
cout << "The Heap is Empty" << endl;
return 0;
}
node *ptr = Find(H1, x);
if (ptr == NULL) {
cout << "Node not found in the Heap" << endl;
return 1;
}
if (ptr->n < k) {
cout << "Entered key greater than current key" << endl;
return 0; mk
}
ptr->n = k;
y = ptr->parent;
if (y != NULL && ptr->n < y->n) {
Cut(H1, ptr, y);
Cascase_cut(H1, y);
}
if (ptr->n < H->n)
H = ptr;
return 0;
}
int FibonacciHeap::Cut(node *H1, node *x, node *y)
{
if (x == x->right)
y->child = NULL;
```

```
(x->left)->right = x->right;
(x->right)->left = x->left;
if (x == y->child)
y->child = x->right;
y->degree = y->degree - 1;
x->right = x;
x->left = x;
(H1->left)->right = x;
x->right = H1;
x->left = H1->left;
H1->left = x;
x->parent = NULL;
x->mark = 'F';
}
int FibonacciHeap::Cascase_cut(node *H1, node *y) {
node *z = y->parent;
if (z != NULL) {
if (y->mark == 'F') {
y->mark = 'T';
} else
{
Cut(H1, y, z);
Cascase_cut(H1, z);
}
}
}
node *FibonacciHeap::Find(node *H, int k) {
node *x = H;
x->C = 'Y';
node *p = NULL;
```

```
if (x->n == k) {
p = x;
x->C = 'N';
return p;
}
if (p == NULL) {
if (x->child != NULL)
p = Find(x->child, k);
if ((x->right)->C != 'Y')
p = Find(x->right, k);
}
x->C = 'N';
return p;
}
int FibonacciHeap::Delete_key(node *H1, int k) {
node *np = NULL;
int t;
t = Decrease_key(H1, k, -5000);
if (!t)
np = Extract_Min(H);
if (np != NULL)
cout << "Key Deleted" << endl;
else
cout << "Key not Deleted" << endl;
return 0;
}
int main() {
int n, m, l;
FibonacciHeap fh;
node *p;
```

```
node *H;
H = fh.InitializeHeap();
p = fh.Create_node(7);
H = fh.Insert(H, p);
p = fh.Create_node(3);
H = fh.Insert(H, p);
p = fh.Create_node(17);
H = fh.Insert(H, p);
p = fh.Create_node(24);
H = fh.Insert(H, p);
fh.Display(H);
p = fh.Extract_Min(H);
if (p != NULL)
cout << "The node with minimum key: " << p->n << endl;
else
cout << "Heap is empty" << endl;
m = 26;
l = 16;
fh.Decrease_key(H, m, l);
m = 16;
fh.Delete_key(H, m);
}
```

OUTPUT:

```
Root Nodes:
3-->7-->17-->24
The node with minimum key: 3
Node not found in the Heap
Node not found in the Heap
Key not Deleted

--------------------------------
Process exited after 0.05072 seconds with return value 0
Press any key to continue . . .
```

Breadth First Search

```
#include <iostream>
#include <list>
using namespace std;
class Graph {
  int numVertices;
  list<int>* adjLists;
  bool* visited;
   public:
  Graph(int vertices);
  void addEdge(int src, int dest);
  void BFS(int startVertex);
};
Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
}
void Graph::addEdge(int src, int dest) {
  adjLists[src].push_back(dest);
  adjLists[dest].push_back(src);
}
void Graph::BFS(int startVertex) {
  visited = new bool[numVertices];
  for (int i = 0; i < numVertices; i++)
    visited[i] = false;
  list<int> queue;
  visited[startVertex] = true;
  queue.push_back(startVertex);
  list<int>::iterator i;
  while (!queue.empty()) {
```

```
    int currVertex = queue.front();
    cout << "Visited " << currVertex << " ";
    queue.pop_front();
    for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {
      int adjVertex = *i;
      if (!visited[adjVertex]) {
        visited[adjVertex] = true;
        queue.push_back(adjVertex);
      }
    }
  }
}
int main() {
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 0);
  g.addEdge(2, 3);
  g.addEdge(3, 3);
  g.BFS(2);
  return 0;
}
```

OUTPUT:

```
Visited 2 Visited 0 Visited 1 Visited 3
--------------------------------
Process exited after 0.05633 seconds with return value 0
Press any key to continue . . .
```

Depth First Search

```
#include <iostream>
#include <list>
using namespace std;
class Graph {
  int numVertices;
  list<int> *adjLists;
  bool *visited;
   public:
  Graph(int V);
  void addEdge(int src, int dest);
  void DFS(int vertex);
};
Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
  visited = new bool[vertices];
}
void Graph::addEdge(int src, int dest) {
  adjLists[src].push_front(dest);
}
void Graph::DFS(int vertex) {
  visited[vertex] = true;
  list<int> adjList = adjLists[vertex];
  cout << vertex << " ";
  list<int>::iterator i;
  for (i = adjList.begin(); i != adjList.end(); ++i)
    if (!visited[*i])
      DFS(*i);
```

```
}
int main() {
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 3);
  g.DFS(2);
  return 0;
}
```

OUTPUT:

```
2 3
--------------------------------
Process exited after 0.04925 seconds with return value 0
Press any key to continue . . .
```

Spanning tree implementation

```
#include <cstring>
#include <iostream>
using namespace std;
#define INF 9999999
#define V 5
int G[V][V] = {
  {0, 9, 75, 0, 0},
  {9, 0, 95, 19, 42},
  {75, 95, 0, 51, 66},
  {0, 19, 51, 0, 31},
  {0, 42, 66, 31, 0}};
int main() {
  int no_edge;
  int selected[V];
  memset(selected, false, sizeof(selected));
  no_edge = 0;
  selected[0] = true;
  int x;
  int y;
  cout << "Edge"
     << " : "
     << "Weight";
  cout << endl;
  while (no_edge < V - 1) {
    int min = INF;
    x = 0;
    y = 0;
    for (int i = 0; i < V; i++) {
```

```
      if (selected[i]) {
        for (int j = 0; j < V; j++) {
          if (!selected[j] && G[i][j]) {
            if (min > G[i][j]) {
              min = G[i][j];
              x = i;
              y = j;
            }
          }
        }
      }
    }
    cout << x << " - " << y << " :  " << G[x][y];
    cout << endl;
    selected[y] = true;
    no_edge++;
  }
  return 0;
}
```

OUTPUT:

```
Edge : Weight
0 - 1 :  9
1 - 3 :  19
3 - 4 :  31
3 - 2 :  51

--------------------------------
Process exited after 0.03622 seconds with return value 0
Press any key to continue . . .
```