**CẤU TRÚC DỮ LIỆU**

**I. Lý thuyết**

**1. Cây nhị phân**

|  |  |
| --- | --- |
| - Cây nhị phân (Binary Tree) là một cấu trúc dữ liệu dạng cây mà mỗi nút có tối đa hai nút con. | - Cây tìm kiếm nhị phân (Binary Search Tree) là một cấu trúc dữ liệu dạng cây mà mỗi nút có tối đa hai nút con, nút con bên trái có giá trị nhỏ hơn nút cha và nút con bên phải có giá trị lớn hơn nút cha. |

**2. Cây tổng quát**

|  |
| --- |
| - Cây tổng quát (General Tree) là một cấu trúc dữ liệu dạng cây mà mỗi nút có thể có nhiều hơn hai nút con. |

**3. Danh sách liên kết đơn**

|  |
| --- |
| - Danh sách liên kết đơn (Singly Linked List) là một cấu trúc dữ liệu động,  trong đó các phần tử được liên kết với nhau theo một thứ tự tuyến tính. |

**4. Ngăn xếp**

|  |
| --- |
| - Ngăn xếp (Stack) là một cấu trúc dữ liệu dạng danh sách mà các phần tử được thêm vào và lấy ra theo nguyên tắc LIFO (Last In First Out). |

**5. Hàng đợi**

|  |
| --- |
| - Hàng đợi (Queue) là một cấu trúc dữ liệu dạng danh sách mà các phần tử được thêm vào ở cuối và lấy ra ở đầu theo nguyên tắc FIFO (First In First Out). |

**II. Code**

**1. Cây nhị phân**

|  |
| --- |
| #include <iostream>  using namespace std;  class TreeNode {  public:      int value;      TreeNode\* left;      TreeNode\* right;      TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}  };  class BinaryTree {  private:      TreeNode\* root;      TreeNode\* addNode(TreeNode\* node, int key) {          if (!node) return new TreeNode(key);          if (key < node->value) node->left = addNode(node->left, key);          else if (key > node->value) node->right = addNode(node->right, key);          return node;      }      TreeNode\* findMin(TreeNode\* node) {          while (node->left) node = node->left;          return node;      }      TreeNode\* deleteNode(TreeNode\* node, int key) {          if (!node) return node;          if (key < node->value) node->left = deleteNode(node->left, key);          else if (key > node->value) node->right = deleteNode(node->right, key);          else {              if (!node->left) {                  TreeNode\* temp = node->right;                  delete node;                  return temp;              } else if (!node->right) {                  TreeNode\* temp = node->left;                  delete node;                  return temp;              }              TreeNode\* temp = findMin(node->right);              node->value = temp->value;              node->right = deleteNode(node->right, temp->value);          }          return node;      }      bool searchNode(TreeNode\* node, int key) {          if (!node) return false;          if (key < node->value) return searchNode(node->left, key);          else if (key > node->value) return searchNode(node->right, key);          return true;      }      void inorder(TreeNode\* node) {          if (node) {              inorder(node->left);              cout << node->value << " ";              inorder(node->right);          }      }      void preorder(TreeNode\* node) {          if (node) {              cout << node->value << " ";              preorder(node->left);              preorder(node->right);          }      }      void postorder(TreeNode\* node) {          if (node) {              postorder(node->left);              postorder(node->right);              cout << node->value << " ";          }      }      int height(TreeNode\* node) {          if (!node) return 0;          return max(height(node->left), height(node->right)) + 1;      }      int findMax(TreeNode\* node) {          while (node->right) node = node->right;          return node->value;      }      int findMinValues(TreeNode\* node) {          while (node->left) node = node->left;          return node->value;      }  public:      BinaryTree() : root(nullptr) {}      void addNode(int key) {          root = addNode(root, key);      }      void deleteNode(int key) {          root = deleteNode(root, key);      }      bool searchNode(int key) {          return searchNode(root, key);      }      void inorder() {          inorder(root);          cout << endl;      }      void preorder() {          preorder(root);          cout << endl;      }      void postorder() {          postorder(root);          cout << endl;      }      int height() {          return height(root);      }      int findMax() {          if (!root) throw runtime\_error("Tree is empty");          return findMax(root);      }      int findMin() {          if (!root) throw runtime\_error("Tree is empty");          return findMinValues(root);      }  }; |

**2. Cây tổng quát**

|  |
| --- |
| #include <iostream>  #include <vector>  template <class N>  class Node {  public:    N value;    std::vector<Node<N>\*> children;    Node(N val) : value(val) {}  };  template <class G>  class GT {  private:    Node<G>\* root;    Node<G>\* insertRecursive(Node<G>\* node, G value) {      if (node == nullptr) {        return new Node<G>(value);      }      node->children.push\_back(new Node<G>(value));      return node;    }    Node<G>\* searchRecursive(Node<G>\* node, G value) {      if (node == nullptr) {        return nullptr;      }      if (node->value == value) {        return node;      }      for (Node<G>\* child : node->children) {        Node<G>\* result = searchRecursive(child, value);        if (result != nullptr) {          return result;        }      }      return nullptr;    }    Node<G>\* deleteRecursive(Node<G>\* node, G value) {      if (node == nullptr) {        return node;      }      if (node->value == value) {        delete node;        return nullptr;      }      for (Node<G>\* child : node->children) {        child = deleteRecursive(child, value);      }      return node;    }    void inorderRecursive(Node<G>\* node) {      if (node == nullptr) {        return;      }      for (Node<G>\* child : node->children) {        inorderRecursive(child);      }      std::cout << node->value << " ";    }    void preorderRecursive(Node<G>\* node) {      if (node == nullptr) {        return;      }      std::cout << node->value << " ";      for (Node<G>\* child : node->children) {        preorderRecursive(child);      }    }    void postorderRecursive(Node<G>\* node) {      if (node == nullptr) {        return;      }      for (Node<G>\* child : node->children) {        postorderRecursive(child);      }      std::cout << node->value << " ";    }  }; |

**3. Danh sách liên kết đơn**

|  |
| --- |
| #include <iostream>  template <class S>  class Node {    public:      S data;      Node\* next;      Node(S data, Node\* next = nullptr) : data(data), next(next) {}  };  template <class S>  class SList {    private:      Node<S>\* head;      Node<S>\* tail;    public:      SList() : head(nullptr), tail(nullptr) {}      ~SList() {        Node<S>\* node;        while (head != nullptr) {          node = head;          head = head->next;          delete node;        }      }      void printList() const {        for (Node<S>\* p = head; p != nullptr; p = p->next) {          std::cout << p->data << std::endl;        }      }      void push\_front(S x) {        head = new Node<S>(x, head);        if (tail == nullptr) {          tail = head;        }      }      void push\_at(S x, int pos) {        Node<S>\*\* pp = &head;        for (int i = 0; i < pos && \*pp; ++i) {          pp = &(\*pp)->next;        }        \*pp = new Node<S>(x, \*pp);        if ((\*pp)->next == nullptr) {          tail = \*pp;        }      }      void push\_back(S x) {        Node<S>\*\* pp = &head;        while (\*pp) {          pp = &(\*pp)->next;        }        \*pp = new Node<S>(x);        tail = \*pp;      }      void erase(S x) {        Node<S>\*\* pp = &head;        while (\*pp && (\*pp)->data != x) {          pp = &(\*pp)->next;        }        if (\*pp) {          Node<S>\* toDelete = \*pp;          \*pp = (\*pp)->next;          if (toDelete == tail) {            tail = \*pp;          }          delete toDelete;        }      }      bool search(S x) const {        for (Node<S>\* p = head; p != nullptr; p = p->next) {          if (p->data == x) {            return true;          }        }        return false;      }  }; |

**4. Ngăn xếp**

|  |
| --- |
| #include <iostream>  class StackSl {    private:      struct Node {        int elem;        Node \*next;      };      Node \*top;    public:      StackSl() {        top = nullptr;      }      ~StackSl() {        while (top != nullptr) {          Node \*tmp = top;          top = top->next;          delete tmp;        }      }      void push(int x) {        Node \*node = new Node;        node->elem = x;        node->next = top;        top = node;      }      int pop() {        Node \*tmp = top;        int x = top->elem;        top = top->next;        delete tmp;        return x;      }      int topElement() {        return top->elem;      }      int size() {        int count = 0;        Node \*tmp = top;        while (tmp != nullptr) {          count++;          tmp = tmp->next;        }        return count;      }      bool isEmpty() {        return top == nullptr;      }      void printStack() {        Node \*tmp = top;        while (tmp != nullptr) {          std::cout << tmp->elem << " ";          tmp = tmp->next;        }        std::cout << std::endl;      }  }; |

**5. Hàng đợi**

|  |
| --- |
| #include <iostream>  template <class N>  class Node {    public:      N data;      Node\* next;      Node(N data, Node\* next = nullptr) : data(data), next(next) {}  };  template <class Q>  class Queue {    private:      Node<Q>\* front;      Node<Q>\* rear;    public:      Queue() : front(nullptr), rear(nullptr) {}      ~Queue() {        Node<Q>\* node;        while (front != nullptr) {          node = front;          front = front->next;          delete node;        }      }      void printQueue() const {        for (Node<Q>\* p = front; p != nullptr; p = p->next) {          std::cout << p->data << std::endl;        }      }      void enqueue(Q x) {        Node<Q>\* node = new Node<Q>(x);        if (front == nullptr) {          front = node;        } else {          rear->next = node;        }        rear = node;      }      Q dequeue() {        Node<Q>\* tmp = front;        Q x = front->data;        front = front->next;        delete tmp;        return x;      }      Q frontElement() {        return front->data;      }      Q rearElement() {        return rear->data;      }      int size() {        int count = 0;        Node<Q>\* tmp = front;        while (tmp != nullptr) {          count++;          tmp = tmp->next;        }        return count;      }      bool isEmpty() {        return front == nullptr;      }  }; |