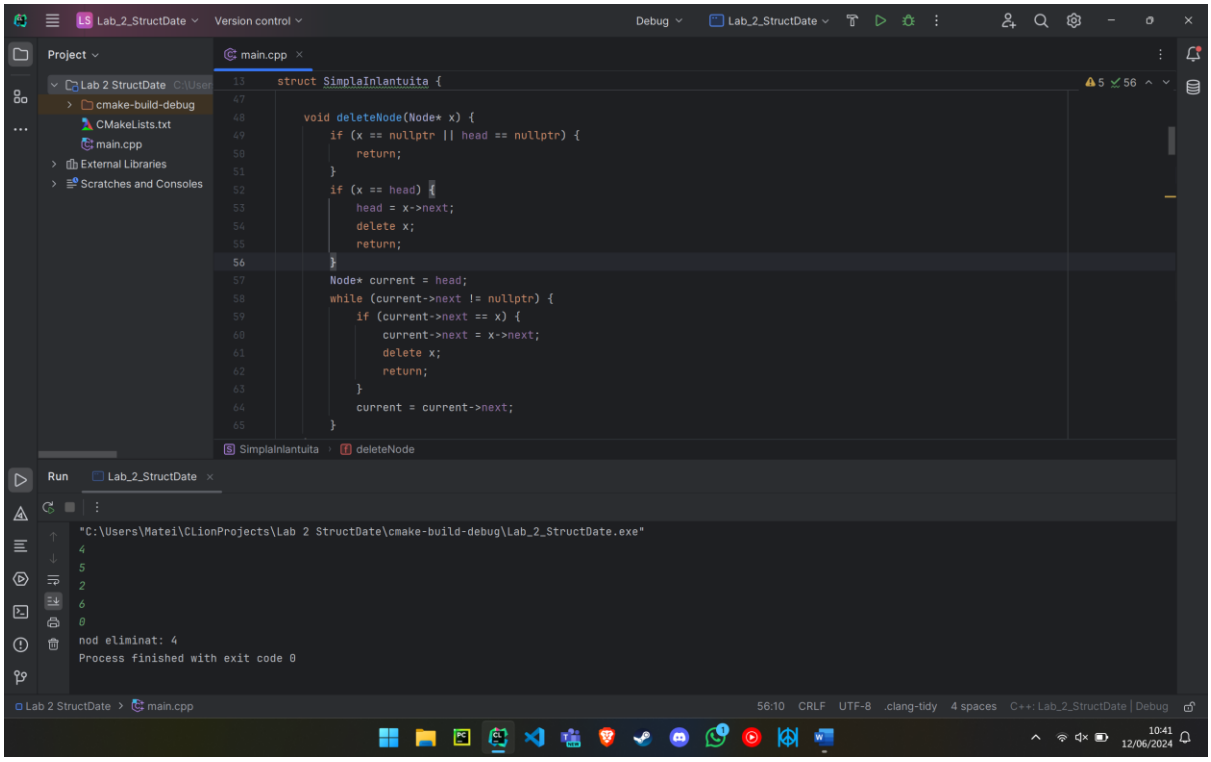
UNGUREANU Matei-Stefan
Grupa 132

Laborator 1:
1.Pasul comun al algoritmilor Selection Sort si Heap Sort este swap()
2.Algoritmul Heap Sort are urmatorii pasi:
- Se creaza o structura de heap
- Se parcurge array-ul de la final la start si se face un swap
- Dupa fiecare swap() se recreaza structura de heap ca sa functioneze algoritmul in continuare

Laborator 2:
1.Codul pentru stergerea capului unei liste circulare dublu inlantuite este:

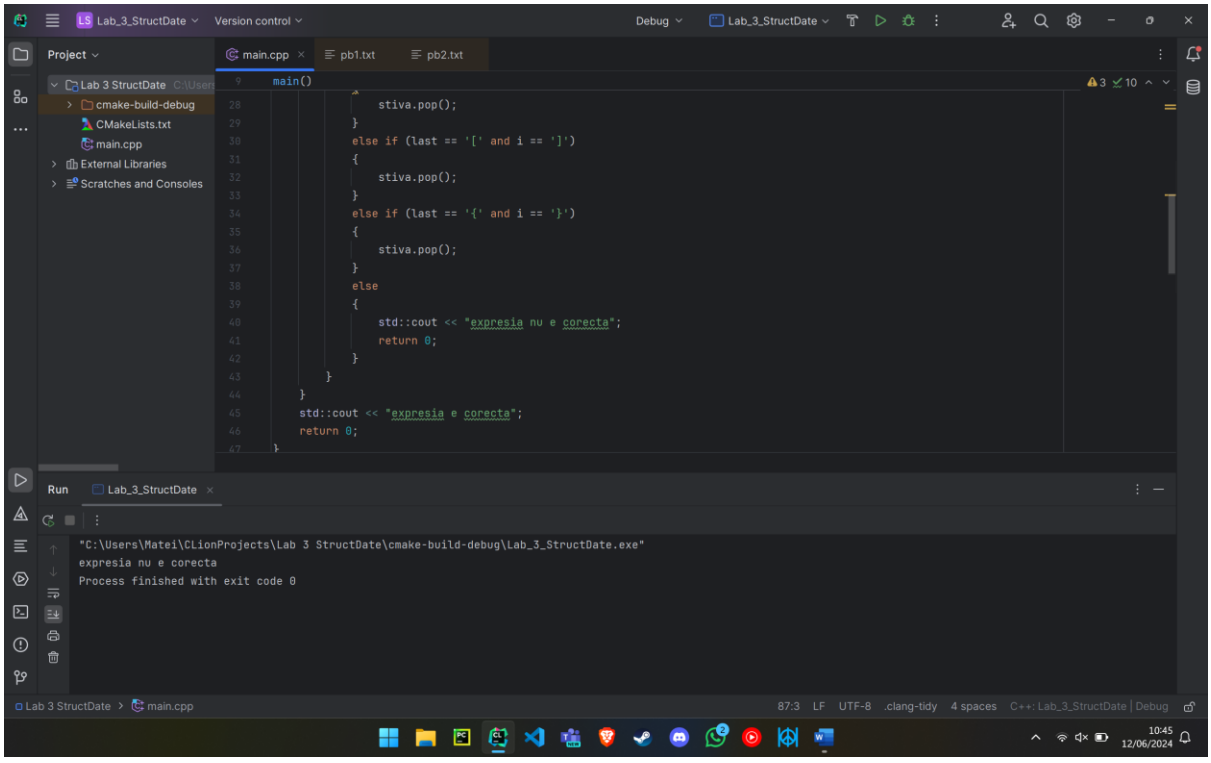
```
void deleteNode(Node* x) {
    if (x == nullptr || head == nullptr) return;

    if (x == head) {
      if (head->next == head) {
        delete head;
        head = nullptr;
      } else {
        head->prev->next = head->next;
        head->next->prev = head->prev;
        Node* temp = head;
        head = head->next;
        delete temp;
      }
    } else {
      x->prev->next = x->next;
      x->next->prev = x->prev;
      delete x;
    }
  }
```

2.



Laborator 3:

1.

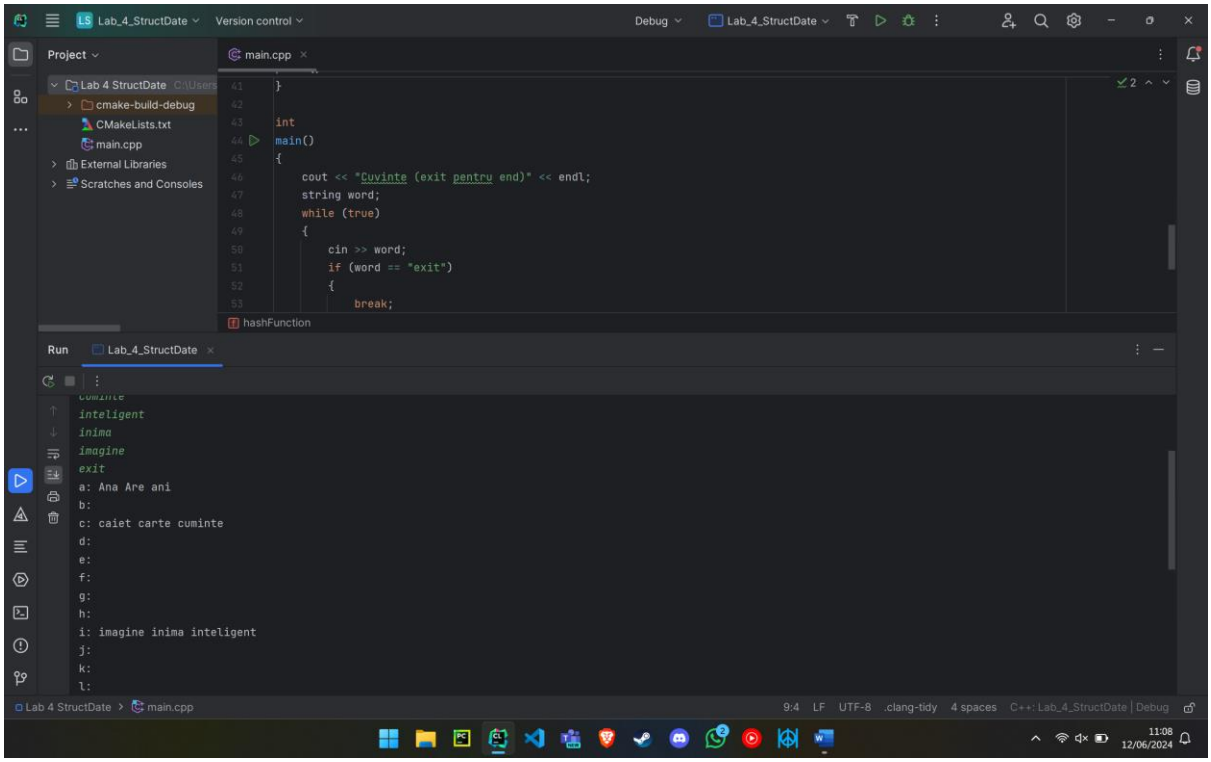


2.Pasii algoritmului pentru rezolvarea pb 3.2 sunt:

- Se citesc numerele si se introduc in vector
- Se parcurge vectorul de la final la start iar in parcurgese se executa rumatoarele:

- Se verifica daca numarul la care am ajuns e mai mare decat capul stivei; daca da, facem pop, daca nu, atunci capul stivei este rezultatul cautat pentru acel numar si se salveaza in alt vector; repetam pasul acesta pana gasim un rezultat cautat
- Numarul e bagat in stiva
- Dupa parcurgere folosim 2 cozi pentru afisare, dand pop la ambele in acelasi timp

Laborator 4:



Laborator 5:

1.Pasii algoritmului pentru rezolvarea problemei 5.1 sunt:
- Stiind ca parcurgerea inordine a unui BST echilibrat este un sir crescator de numere, facem o parcurgere prin backtracking asemanatoare cu cautarea binara
- Citim numerele si le punem in vector
- Gasim mijlocul vectorului – radacina BST-ului
- Apelam aceeasi procedura pentru nodul din stanga al radacinii si pentru nodul din dreapta al radacinii, punandu-le la dispozitie parti din vector corespunzatoare: pentru nod stanga, prima jumatate a vectorului primit, pentru nod dreapta, a doua jumatate a vectorului primit

2.Pasii algoritmului pentru rezolvarea problemei 5.2 sunt:
- Citim arborii
- Obtinem apelam traversarea inordine pentru ambii arbori
- Concatenam si sortam cele 2 traversari inordine, obtinand astfel traversarea inordine a arborelui mare
- Folosim problema 5.1 si obtinem arborele cautat

Laborator6:

1.Diferenta din constructia MST dintre algoritmul lui Kruskal si algoritmul lui Prim este ca Prim porneste de la un nod si mareste MST-ul putin cate putin, nod cu nod, cautand mereu cea mai

“usoara” muchie al carui capat e un nod deja existent in MST si adaugand apoi nodul celalalt in MST, in vreme ce Kruskal porneste de la cea mai “usoara” greutate existenta si formeaza “grupari” de noduri care respecta proprietatile MST, practic mini-MST-uri, care apoi ajung sa se uneasca in timp, formand in final un singur MST

2.Pasii algoritmului lui Prim sunt:

- Se alege un nod la intamplare
- Se cauta cea mai “usoara” muchie care il are la un capat pe nodul ales; nodul de la capatul opus este introdus in MST
- Se repeta pasul de dinainte pana cand toate nodurile sunt in MST

Codul sursa:

Observatie: Pentru laboratoarele in care sunt mai multe probleme am pus comentariu (exemplu: //pb1) inaintea partii de cod care e specifica unei singure probleme. Inainte de comentariu mai pot fi functii care sunt comune ambelor probleme. Pentru a rula un exercitiu, puneti in comentariu cealalta problema incepand de la comentariul //pbX pana la sfarsitul functiei main a problemei respective. (O sa pun textul asta si la sfarsit in caz ca incepeti cu ultimul laborator)

Laborator 1:

```
#include <iostream>
#include <vector>

void printeaza(const std::vector<int>& v) {
    for (int i : v) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
}

// pb1
void selectionSort(std::vector<int>& v) {
    int n = (int)v.size();
    for (int i = 0; i < n - 1; ++i) {
        int minIndex = i;
        for (int j = i + 1; j < n; ++j) {
            if (v[j] < v[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(v[i], v[minIndex]);
    }
}

int main() {
    std::vector<int> v = {64, 25, 12, 22, 11};
    std::cout << "Vector original: ";
    printeaza(v);
```

```
    selectionSort(v);
    std::cout << "Vector sortat: ";
    printeaza(v);
    return 0;
}

// pb2
/*
void heapify(std::vector<int>& v, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && v[left] > v[largest]) {
        largest = left;
    }
    if (right < n && v[right] > v[largest]) {
        largest = right;
    }
    if (largest != i) {
        std::swap(v[i], v[largest]);
        heapify(v, n, largest);
    }
}

void buildHeap(std::vector<int>& v, int n) {
    for (int i = 0; i < n; ++i) {
        int current = i;
        while (current > 0) {
            int parent = (current - 1) / 2;
            if (v[current] > v[parent]) {
                std::swap(v[current], v[parent]);
                current = parent;
            } else {
                break;
            }
        }
    }
}

void heapSort(std::vector<int>& v) {
    int n = (int)v.size();
    buildHeap(v, n);
    for (int i = n - 1; i > 0; --i) {
        std::swap(v[0], v[i]);
        heapify(v, i, 0);
    }
}
```

```
int main() {
   std::vector<int> v = {12, 11, 13, 5, 6, 7};
   std::cout << "Vector original: ";
   printArray(v);
   heapSort(v);
   std::cout << "Vector sortat: ";
   printArray(v);
   return 0;
}
*/
```

Laborator 2:

```
#include <iostream>
#include <string>

using namespace std;


// pb1
struct Node {
   int key;
   Node* next;
};

struct SimplaInlantuita {
   Node* head;

   SimplaInlantuita() : head(nullptr) {}

   ~SimplaInlantuita() {
      Node* current = head;
      while (current != nullptr) {
         Node* temp = current;
         current = current->next;
         delete temp;
      }
   }

   [[nodiscard]] Node* search(int k) const {
      Node* current = head;
      while (current != nullptr) {
         if (current->key == k) {
            return current;
         }
         current = current->next;
      }
      return nullptr;
   }
```

```
void insert(Node* x) {
    if (head == nullptr) {
        head = x;
        x->next = nullptr;
    } else {
        x->next = head;
        head = x;
    }
}

void deleteNode(Node* x) {
    if (x == nullptr || head == nullptr) {
        return;
    }
    if (x == head) {
        head = x->next;
        delete x;
        return;
    }
    Node* current = head;
    while (current->next != nullptr) {
        if (current->next == x) {
            current->next = x->next;
            delete x;
            return;
        }
        current = current->next;
    }
}

[[nodiscard]] Node* minimum() const {
    if (head == nullptr) {
        return nullptr;
    }
    Node* minNode = head;
    Node* current = head->next;
    while (current != nullptr) {
        if (current->key < minNode->key) {
            minNode = current;
        }
        current = current->next;
    }
    return minNode;
}

[[nodiscard]] Node* maximum() const {
    if (head == nullptr) {
```

```
        return nullptr;
    }
    Node* maxNode = head;
    Node* current = head->next;
    while (current != nullptr) {
        if (current->key > maxNode->key) {
            maxNode = current;
        }
        current = current->next;
    }
    return maxNode;
}

static Node* successor(Node* x) {
    return x != nullptr ? x->next : nullptr;
}

Node* predecessor(Node* x) const {
    if (x == nullptr || x == head) {
        return nullptr;
    }

    Node* current = head;
    while (current != nullptr && current->next != x) {
        current = current->next;
    }
    return current;
}
};

int main()
{
    SimplaInlantuita linkedList;
    bool ok = true;
    int x, primul = 0;
    while (ok)
    {
        std::cin >> x;
        if (x == 0)
        {
            ok = false;
        }
        else
        {
            if (primul == 0)
            {
                primul = x;
            }
```

```
            linkedList.insert(new Node{x, nullptr});
        }
    }
    Node *node = linkedList.search(x);
    while(node != nullptr)
    {
        std::cout<<node->key<<" ";
        node = node->next;
    }
    linkedList.deleteNode(node);
    std::cout << "nod eliminat: " << primul;
    return 0;
}
///// am comentat liniile astea ca sa modific codul pentru problema 2 din test
//    linkedList.insert(new Node{5, nullptr});
//    linkedList.insert(new Node{10, nullptr});
//    linkedList.insert(new Node{3, nullptr});
//    linkedList.insert(new Node{8, nullptr});
//
//    Node* node = linkedList.search(3);
//    cout << "Search result: " << (node ? to_string(node->key) : "NIL") << endl;
//
//    Node* minNode = linkedList.minimum();
//    Node* maxNode = linkedList.maximum();
//    cout << "Minimum: " << (minNode ? to_string(minNode->key) : "NIL") << endl;
//    cout << "Maximum: " << (maxNode ? to_string(maxNode->key) : "NIL") << endl;
//
//    Node* successorNode = SimplaInlantuita::successor(node);
//    Node* predecessorNode = linkedList.predecessor(node);
//    cout << "Successor: " << (successorNode ? to_string(successorNode->key) : "NIL") << endl;
//    cout << "Predecessor: " << (predecessorNode ? to_string(predecessorNode->key) : "NIL") <<
endl;
//
//    linkedList.deleteNode(node);
//
//    return 0;
//}


/*
// pb2
struct Node {
    int key;
    Node* prev;
    Node* next;
};

struct DublaInlantuita {
```

```
Node* head;
Node* tail;

DublaInlantuita() : head(nullptr), tail(nullptr) {}

~DublaInlantuita() {
    Node* current = head;
    while (current != nullptr) {
        Node* temp = current;
        current = current->next;
        delete temp;
    }
}

[[nodiscard]] Node* search(int k) const {
    Node* current = head;
    while (current != nullptr) {
        if (current->key == k) {
            return current;
        }
        current = current->next;
    }
    return nullptr;
}

void insert(Node* x) {
    if (head == nullptr) {
        head = x;
        tail = x;
        x->prev = nullptr;
        x->next = nullptr;
    } else {
        x->next = head;
        head->prev = x;
        head = x;
        x->prev = nullptr;
    }
}

void deleteNode(Node* x) {
    if (x == nullptr) {
        return;
    }
    if (x == head) {
        head = x->next;
    }
    if (x == tail) {
        tail = x->prev;
```

```
        }
        if (x->prev != nullptr) {
            x->prev->next = x->next;
        }
        if (x->next != nullptr) {
            x->next->prev = x->prev;
        }
        delete x;
    }

    [[nodiscard]] Node* minimum() const {
        if (head == nullptr)
            return nullptr;

        Node* minNode = head;
        Node* current = head->next;
        while (current != head && current != nullptr) {
            if (current->key < minNode->key)
                minNode = current;
            current = current->next;
        }
        return minNode;
    }

    [[nodiscard]] Node* maximum() const {
        if (head == nullptr)
            return nullptr;

        Node* maxNode = head;
        Node* current = head->next;
        while (current != head && current != nullptr) {
            if (current->key > maxNode->key)
                maxNode = current;
            current = current->next;
        }
        return maxNode;
    }

    static Node* successor(Node* x) {
        return x != nullptr ? x->next : nullptr;
    }

    static Node* predecessor(Node* x) {
        return x != nullptr ? x->prev : nullptr;
    }
};

int main() {
```

```
    DublaInlantuita linkedList;

    linkedList.insert(new Node{5, nullptr, nullptr});
    linkedList.insert(new Node{10, nullptr, nullptr});
    linkedList.insert(new Node{3, nullptr, nullptr});
    linkedList.insert(new Node{8, nullptr, nullptr});

    Node* node = linkedList.search(3);
    cout << "Search result: " << (node ? to_string(node->key) : "NIL") << endl;

    Node* minNode = linkedList.minimum();
    Node* maxNode = linkedList.maximum();
    cout << "Minimum: " << (minNode ? to_string(minNode->key) : "NIL") << endl;
    cout << "Maximum: " << (maxNode ? to_string(maxNode->key) : "NIL") << endl;

    Node* successorNode = DublaInlantuita::successor(node);
    Node* predecessorNode = DublaInlantuita::predecessor(node);
    cout << "Successor: " << (successorNode ? to_string(successorNode->key) : "NIL") << endl;
    cout << "Predecessor: " << (predecessorNode ? to_string(predecessorNode->key) : "NIL") << endl;

    linkedList.deleteNode(node);

    return 0;
}
*/

/*
// pb3
struct Node {
    int key;
    Node* prev;
    Node* next;

    explicit Node(int k) : key(k), prev(nullptr), next(nullptr) {}
};

struct CircularInlanuita {
    Node* head;

    CircularInlanuita() : head(nullptr) {}

    ~CircularInlanuita() {
        if (head == nullptr) return;

        Node* current = head;
        do {
            Node* temp = current;
            current = current->next;
```

```
        delete temp;
    } while (current != head);
}

[[nodiscard]] Node* search(int k) const {
    if (head == nullptr) return nullptr;

    Node* current = head;
    do {
        if (current->key == k) return current;
        current = current->next;
    } while (current != head);

    return nullptr;
}

void insert(Node* x) {
    if (head == nullptr) {
        head = x;
        head->next = head;
        head->prev = head;
    } else {
        x->next = head;
        x->prev = head->prev;
        head->prev->next = x;
        head->prev = x;
        head = x;
    }
}

void deleteNode(Node* x) {
    if (x == nullptr || head == nullptr) return;

    if (x == head) {
        if (head->next == head) {
            delete head;
            head = nullptr;
        } else {
            head->prev->next = head->next;
            head->next->prev = head->prev;
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    } else {
        x->prev->next = x->next;
        x->next->prev = x->prev;
        delete x;
```

```
        }
    }

    [[nodiscard]] Node* minimum() const {
        if (head == nullptr) return nullptr;

        Node* minNode = head;
        Node* current = head->next;
        do {
            if (current->key < minNode->key) minNode = current;
            current = current->next;
        } while (current != head);

        return minNode;
    }

    [[nodiscard]] Node* maximum() const {
        if (head == nullptr) return nullptr;

        Node* maxNode = head;
        Node* current = head->next;
        do {
            if (current->key > maxNode->key) maxNode = current;
            current = current->next;
        } while (current != head);

        return maxNode;
    }

    Node* successor(Node* x) const {
        if (head == nullptr || x == nullptr) return nullptr;
        return x->next;
    }

    Node* predecessor(Node* x) const {
        if (head == nullptr || x == nullptr) return nullptr;
        return x->prev;
    }
};

int main() {
    CircularInlanuita linkedList;

    linkedList.insert(new Node(5));
    linkedList.insert(new Node(10));
    linkedList.insert(new Node(3));
    linkedList.insert(new Node(8));
```

```
    Node* node = linkedList.search(3);
    cout << "Search result: " << (node ? to_string(node->key) : "NIL") << endl;

    Node* minNode = linkedList.minimum();
    Node* maxNode = linkedList.maximum();
    cout << "Minimum: " << (minNode ? to_string(minNode->key) : "NIL") << endl;
    cout << "Maximum: " << (maxNode ? to_string(maxNode->key) : "NIL") << endl;

    Node* successorNode = linkedList.successor(node);
    Node* predecessorNode = linkedList.predecessor(node);
    cout << "Successor: " << (successorNode ? to_string(successorNode->key) : "NIL") << endl;
    cout << "Predecessor: " << (predecessorNode ? to_string(predecessorNode->key) : "NIL") << endl;

    linkedList.deleteNode(node);

    return 0;
}
*/
```

Laborator 3:

```
#include <iostream>
#include <fstream>
#include <stack>
#include <queue>
#include <vector>

/*
// pb1
int
main()
{
    std::stack<char> stiva;
    std::ifstream f("pb1.txt");
    std::string expresie;
    char last;
    std::getline(f, expresie);
//    std::cout<<expresie;
    for (char i : expresie)
    {
        if (i == '(' or i == '[' or i == '{')
        {
            stiva.push(i);
        }
        if (i == ')' or i == ']' or i == '}')
        {
            last = stiva.top();
            if (last == '(' and i == ')')
            {
```

```
            stiva.pop();
        }
        else if (last == '[' and i == ']')
        {
            stiva.pop();
        }
        else if (last == '{' and i == '}')
        {
            stiva.pop();
        }
        else
        {
            std::cout << "expresia nu e corecta";
            return 0;
        }
    }
  }
  std::cout << "expresia e corecta";
  return 0;
}
*/


// pb2
int
main() {
  std::ifstream f("pb2.txt");
  int n;
  f>>n;
  std::vector<int> nums(n);
  for (int i = 0; i < n; ++i) {
    f >> nums[i];
  }
  std::stack<int> s;
  std::queue<int> q1;
  std::queue<int> q2;
  std::vector<int> result(n, -1);
  for (int i = n - 1; i >= 0; --i) {
    while (!s.empty() && nums[s.top()] <= nums[i]) {
      s.pop();
    }
    if (!s.empty()) {
      result[i] = s.top();
    }
    s.push(i);
  }
  for (int i = 0; i < n; ++i) {
    if (result[i] != -1) {
```

```
            q1.push(result[i]);
            q2.push(i);
        }
    }
    while (!q1.empty()) {
        std::cout << q2.front() << " " << q1.front() << "\n";
        q2.pop();
        q1.pop();
    }
    return 0;
}
```

Laborator 4:

```
#include <iostream>
#include <vector>
#include <list>

using namespace std;

vector<list<string>> hashTable(26);

int
hashFunction(const string &word)
{
    char firstLetter = static_cast<char>(tolower(word[0]));
    return firstLetter - 'a';
}

void
insert(const string &word)
{
    int index = hashFunction(word);
    list<string> &bucket = hashTable[index];
    auto it = bucket.begin();
    while (it != bucket.end() && *it < word)
    {
        ++it;
    }
    bucket.insert(it, word);
}

void
print()
{
    for (int i = 0; i < 26; ++i)
    {
        cout << (char)('a' + i) << ": ";
        for (auto &it : hashTable[i])
```

```
        {
            cout << it << " ";
        }
        cout << endl;
    }
}

int
main()
{
    cout << "Cuvinte (exit pentru end)" << endl;
    string word;
    while (true)
    {
        cin >> word;
        if (word == "exit")
        {
            break;
        }
        insert(word);
    }
    print();
    return 0;
}
```

Laborator 5:

```
#include <iostream>
#include <vector>

using namespace std;

struct TreeNode
{
    int val;
    TreeNode *left;
    TreeNode *right;
    explicit TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

void
deleteTree(TreeNode *root)
{
    if (root)
    {
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
```

```
}

TreeNode *
sortedArrayToBST(vector<int> &nums, int start, int end)
{
   if (start > end)
      return nullptr;
   int mid = start + (end - start) / 2;
   auto *root = new TreeNode(nums[mid]);
   root->left = sortedArrayToBST(nums, start, mid - 1);
   root->right = sortedArrayToBST(nums, mid + 1, end);
   return root;
}

void
inorderPrint(TreeNode *root)
{
   if (root)
   {
      inorderPrint(root->left);
      cout << root->val << " ";
      inorderPrint(root->right);
   }
}

/*
// pb1
void
preorderPrint(TreeNode *root)
{
   if (root)
   {
      cout << root->val << " ";
      preorderPrint(root->left);
      preorderPrint(root->right);
   }
}

void
postorderPrint(TreeNode *root)
{
   if (root)
   {
      postorderPrint(root->left);
      postorderPrint(root->right);
      cout << root->val << " ";
   }
}
```

```
int
main()
{
   vector<int> sorted_array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
   TreeNode *bst_root = sortedArrayToBST(sorted_array, 0, (int)sorted_array.size() - 1);
   inorderPrint(bst_root);
   cout << endl;
   preorderPrint(bst_root);
   cout << endl;
   postorderPrint(bst_root);
   deleteTree(bst_root);
   return 0;
}
*/


// pb 2
void inorderTraversal(TreeNode* root, vector<int>& result) {
   if (root == nullptr) return;
   inorderTraversal(root->left, result);
   result.push_back(root->val);
   inorderTraversal(root->right, result);
}

vector<int> mergeArrays(const vector<int>& arr1, const vector<int>& arr2) {
   vector<int> merged;
   int i = 0, j = 0;
   while (i < arr1.size() && j < arr2.size()) {
      if (arr1[i] < arr2[j]) {
         merged.push_back(arr1[i]);
         i++;
      } else {
         merged.push_back(arr2[j]);
         j++;
      }
   }
   while (i < arr1.size()) {
      merged.push_back(arr1[i]);
      i++;
   }
   while (j < arr2.size()) {
      merged.push_back(arr2[j]);
      j++;
   }
   return merged;
}
```

```
TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
   vector<int> array1, array2;
   inorderTraversal(root1, array1);
   inorderTraversal(root2, array2);
   vector<int> mergedArray = mergeArrays(array1, array2);
   return sortedArrayToBST(mergedArray, 0, (int)mergedArray.size() - 1);
}

int main() {
   auto* root1 = new TreeNode(3);
   root1->left = new TreeNode(1);
   root1->right = new TreeNode(5);
   auto* root2 = new TreeNode(4);
   root2->left = new TreeNode(2);
   root2->right = new TreeNode(6);
   TreeNode* mergedTree = mergeTrees(root1, root2);
   inorderPrint(mergedTree);
   cout << endl;
   deleteTree(root1);
   deleteTree(root2);
   return 0;
}
```

Laborator 6:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;


// pb1 Kruskal
struct Edge
{
   int src, dest, weight;
};

bool
compareEdges(const Edge &a, const Edge &b)
{
   return a.weight < b.weight;
}

class UnionFind
{
public:
   explicit UnionFind(int n) : parent(n), rank(n, 0)
   {
```

```
    for (int i = 0; i < n; ++i)
        parent[i] = i;
  }

  int
  find(int u)
  {
    if (u != parent[u])
        parent[u] = find(parent[u]);
    return parent[u];
  }

  void
  unite(int u, int v)
  {
    int rootU = find(u);
    int rootV = find(v);
    if (rootU != rootV)
    {
        if (rank[rootU] > rank[rootV])
            parent[rootV] = rootU;
        else if (rank[rootU] < rank[rootV])
            parent[rootU] = rootV;
        else
        {
            parent[rootV] = rootU;
            rank[rootU]++;
        }
    }
  }

private:
  vector<int> parent;
  vector<int> rank;
};

vector<Edge>
kruskalMST(int V, vector<Edge> &edges)
{
  sort(edges.begin(), edges.end(), compareEdges);
  UnionFind uf(V);
  vector<Edge> mst;
  for (const auto &edge : edges)
  {
    if (uf.find(edge.src) != uf.find(edge.dest))
    {
        mst.push_back(edge);
        uf.unite(edge.src, edge.dest);
```

```
        }
    }
    return mst;
}

int
main()
{
    int V = 4;
    vector<Edge> edges = {
        {0, 1, 1},
        {0, 2, 2},
        {1, 2, 3},
        {1, 3, 4},
        {2, 3, 5}
    };
    vector<Edge> mst = kruskalMST(V, edges);
    cout << "Edges in the Minimum Spanning Tree:" << endl;
    for (const auto &edge : mst)
    {
        cout << edge.src << " -- " << edge.dest << " == " << edge.weight << endl;
    }
    return 0;
}

/*
// pb2 Prim
#include <queue>
#include <utility>

void primMST(const vector<vector<pair<int, int>>>& graph, int V) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    vector<int> key(V, INT_MAX);
    vector<int> parent(V, -1);
    vector<bool> inMST(V, false);
    int startVertex = 0;
    pq.emplace(0, startVertex);
    key[startVertex] = 0;
    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        if (inMST[u])
            continue;
        inMST[u] = true;
        for (const auto& [weight, v] : graph[u]) {
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                pq.emplace(key[v], v);
```

```
            parent[v] = u;
        }
    }
  }
  cout << "Edges in the Minimum Spanning Tree:" << endl;
  for (int i = 1; i < V; ++i) {
    cout << parent[i] << " -- " << i << " == " << key[i] << endl;
  }
}

int main() {
  int V = 4;
  vector<vector<pair<int, int>>> graph(V);
  graph[0].emplace_back(1, 1);
  graph[0].emplace_back(2, 2);
  graph[1].emplace_back(1, 0);
  graph[1].emplace_back(3, 2);
  graph[1].emplace_back(4, 3);
  graph[2].emplace_back(2, 0);
  graph[2].emplace_back(3, 1);
  graph[2].emplace_back(5, 3);
  graph[3].emplace_back(4, 1);
  graph[3].emplace_back(5, 2);
  primMST(graph, V);
  return 0;
}
*/
```

Observatie: Pentru laboratoarele in care sunt mai multe probleme am pus comentariu (exemplu: //pb1) inaintea partii de cod care e specifica unei singure probleme. Inainte de comentariu mai pot fi functii care sunt comune ambelor probleme. Pentru a rula un exercitiu, puneti in comentariu cealalta problema incepand de la comentariul //pbX pana la sfarsitul functiei main a problemei respective. (O sa pun textul asta si la inceput in caz ca incepeti cu primul laborator)