Государственный Университет Молдовы

Факультет Математики и Информатики

Кафедра Информатики

**Лабораторная работа №3**

по курсу «Алгоритмы и Структуры Данных»

Динамические структуры данных.

Выполнил студент группы I2102:

**Король Владимир**

Проверил преподаватель:

**Кроитор Михаил**

Реализуйте следующие операции:

* обход и вывод,
* вставку,
* поиск,
* удаление,

для следующих динамических структур данных (на выбор):

* односвязные списки,
* двусвязные списки,
* кольцевые списки,
* стеки,
* очереди.

Реализовать бинарное дерево поиска и следующие операции:

* обход в прямом порядке,
* обход в обратном порядке,
* центрированный/симметричный обход,
* вставку элемента,
* удаление элемента,
* поиск элемента.

Односвязный список – это динамическая структура данных, состоящая из узлов. Каждый узел будет иметь какие-то значение и указатель на следующий узел.

template<typename T>

struct ListOne {

T data;

ListOne\* next;

};

**Отображение элемента**

void iterate(iteratorCallback<T> callback) {

ListOne<T>\* curPtr = strPtr;

while (curPtr != nullptr) {

callback(&(curPtr->data));

curPtr = curPtr->next;

}

}

**Вставка элемента**

*Для вставки элемента в начало, в новом элементе следует указать ссылку на начальный элемент текущего списка:*

**Вставка в начало:**

ListOne<T>\* prepend(T& data) {

ListOne<T>\* newPtr = new ListOne<T>;

if (newPtr == nullptr)

return nullptr;

newPtr->data = data;

newPtr->next = strPtr;

strPtr = newPtr;

return newPtr;

}

**Вставка в позицию**

ListOne<T>\* insert(ListOne<T>\* after, T& data) {

ListOne<T>\* newPtr = new ListOne<T>;

newPtr->data = data;

ListOne<T>\* currentPtr = after->next;

newPtr->next = currentPtr;

after->next = newPtr;

return newPtr;

}

**Вставка в конец**

ListOne<T>\* append(T& data) {

ListOne<T>\* newPtr = new ListOne<T>;

if (newPtr == nullptr)

return nullptr;

newPtr->data = data;

ListOne<T>\* curPtr = strPtr;

ListOne<T>\* nextPtr = curPtr->next;

while (nextPtr != nullptr) {

curPtr = nextPtr;

nextPtr = nextPtr->next;

}

currentPtr->next = newPtr;

newPtr->next = nullptr;

return newPtr;

}

**Удаление**

*Для удаления элемента из списка, следует разорвать связи, состоящие до и после элемента, после чего удалить указатель на текущий элемент:*

void deleteNode(T& data) {

if (strPtr == nullptr)

return;

ListOne<T>\* curPtr = nullptr, \* previousPtr = nullptr, \* tempPtr = nullptr;

if (strPtr->data == data) {

tempPtr = strPtr;

strPtr = strPtr->next;

delete tempPtr;

return;

}

curPtr = strPtr->next;

previousPtr = strPtr;

while (curPtr != nullptr) {

if (curPtr->data == data) {

tempPtr = curPtr;

previousPtr->next = curPtr->next;

delete tempPtr;

return;

}

previousPtr = curPtr;

curPtr = curPtr->next;

}

}

Двусвязный список – отличием двусвязного списка от односвязного списка состоит в том, что в каждом узле двусвязного списка также имеется ссылка на предыдущий элемент

template <typename T>

struct ListDouble {

T data;

ListDouble\* last;

ListDouble\* next;

};

**Отображение элемента**

void iterateForward(iteratorCallback<T> callback) {

ListDouble<T>\* current = head;

while (current != nullptr) {

callback(&(current->data));

current = current->next;

}

}

Можно заметить, что для **вставки** элемента, придется также указать ссылку на предыдущий элемент

ListDouble<T>\* prepend(T item) {

ListDouble<T>\* newPtr = new ListDouble<T>;

if (newPtr == nullptr)

return nullptr;

newPtr->data = item;

newPtr->next = head;

newPtr->last = nullptr;

if (head != nullptr)

head->last = newPtr;

head = newPtr;

if (tail == nullptr)

tail = head;

return newPtr;

}

Также при **удалении** следует разорвать связь с предыдущим элементом:

void deleteNode(ListDouble<T>\* item) {

if (head == item) {

head = item->next;

if (head)

head->last = nullptr;

else tail = nullptr;

}

else if (tail == item) {

tail = item->last;

if (tail)

tail->next = nullptr;

else head = nullptr;

}

else {

ListDouble<T>\* next = item->next;

ListDouble<T>\* last = item->last;

next->prev = last;

last->next = next;

delete item;

}

}

};

Поскольку у данного списка известны и начало и конец, то итерация может происходить как из начала в конец, так и наоборот:

void iterateForward(iteratorCallback<T> callback) {

ListDouble<T>\* current = head;

while (current != nullptr) {

callback(&(current->data));

current = current->next;

}

}

void iterateBackward(iteratorCallback<T> callback) {

ListDouble<T>\* current = tail;

while (current != nullptr) {

callback(&(current->data));

current = current->last;

}

}

Аналогичную ситуацию можно наблюдать и для **поиска**:

template <typename K>

T\* searchBackward(K& item) {

ListDouble<T>\* current = tail;

while (current != nullptr) {

if (current->data == item)

return &(current->data);

current = current->last;

}

return nullptr;

}

template <typename K>

T\* searchForward(K& item) {

ListDouble<T>\* current = head;

while (current != nullptr) {

if (current->data == item)

return &(current->data);

current = current->next;

}

return nullptr;

}

Кольцевой поиск – в отличии от двусвязного списка, кольцевой поиск будет иметь указатели на следующий элемент конца и предыдущий элемент начала. Единственным отличием при всех итерациях будет то, что цикл будет выполняться до тех пор, пока список не будет пройден полностью, а не до тех пор, пока не будет существовать указатель.

template <typename T>

struct ListCircle {

T data;

ListCircle\* last;

ListCircle\* next;

};

**Отображение**

void iterateForward(CListNode<T>\* start, iteratorCallback<T> callback) {

ListCircle<T>\* current = start;

if (current == nullptr)

return;

do {

callback(&(current->data));

current = current->next;

} while (current != start);

}

**Вставка в начало**

ListCircle<T>\* prepend(T item) {

ListCircle<T>\* newPtr = new ListCircle<T>;

if (newPtr == nullptr)

return nullptr;

newPtr->data = item;

newPtr->next = head;

newPtr->last = nullptr;

if (head != nullptr)

head->last = newPtr;

head = newPtr;

if (tail == nullptr)

tail = head;

updateLoop();

return newPtr;

}

**Вставка в конец**

ListCircle<T>\* append(T item) {

ListCircle<T>\* newPtr = new ListCircle<T>;

if (newPtr == nullptr)

return nullptr;

newPtr->data = item;

newPtr->next = nullptr;

newPtr->last = tail;

if (tail != nullptr)

tail->next = newPtr;

tail = newPtr;

if (head == nullptr)

head = tail;

updateLoop();

return newPtr;

}

**Вставка в определенную позицию**

ListCircle<T>\* insertBefore(PListCircle<T>& source, T item) {

if (source->last == nullptr)

return prepend(item);

ListCircle<T>\* newPtr = new ListCircle<T>;

if (newPtr == nullptr)

return nullptr;

newPtr->data = item;

ListCircle<T>\* temp = source->last;

source->last = newPtr;

newPtr->next = source;

temp->next = newPtr;

newPtr->last = temp;

updateLoop();

return newPtr;

}

ListCircle<T>\* insertAfter(PListCircle<T>& source, T item) {

if (source->next == nullptr)

return append(item);

ListCircle<T>\* newPtr = new ListCircle<T>;

if (newPtr == nullptr)

return nullptr;

newPtr->data = item;

ListCircle<T>\* temp = source->next;

source->next = newPtr;

newPtr->last = source;

temp->last = newPtr;

newPtr->next = temp;

updateLoop();

return newPtr;

}

**Удаление**

void deleteNode(CListNode<T>\* item) {

if (head == item) {

head = item->next;

if (head)

head->last = nullptr;

else tail = nullptr;

}

else if (tail == item) {

tail = item->last;

if (tail)

tail->next = nullptr;

else head = nullptr;

}

else {

ListCircle<T>\* next = item->next;

ListCircle<T>\* last = item->last;

next->last = last;

last->next = next;

delete item;

}

updateLoop();

}

Также, при изменении структуры списка, будет вызываться метод, который будет обновлять ссылки концов списка:

void updateLoop() {

if (head != nullptr && tail != nullptr) {

head->last = tail;

tail->next = head;

}

}

Стек – стек работает по принципу “последний пришел, первый ушел”. Структура, описывающая элемент стека, выглядит аналогично списку:

template<typename T>

struct StackNode {

T data;

struct StackNode\* nextPtr;

};

Отличием стека и очереди от списка является поведение метода **добавления** и **удаления** элемента:

void push(T data) {

StackNode<T>\* newPtr = new StackNode<T>;

if (newPtr == nullptr)

return;

newPtr->data = data;

newPtr->nextPtr = top;

top = newPtr;

}

T pop() {

if (top == nullptr)

return NULL;

T temp = top->data;

StackNode<T>\* ptop = top;

top = top->nextPtr;

delete ptop;

return temp;

}

T peek() {

return \*top->data;

}

**Итерация** и **поиск** будет происходить по всем элементам до тех пор, пока не дойдет до последнего или не найдет необходимый элемент

void iterate(iteratorCallback<T> callback) {

StackNode<T>\* currentPtr = top;

while (currentPtr != nullptr) {

callback(&(currentPtr->data));

currentPtr = currentPtr->nextPtr;

}

}

T\* search(K &item) {

StackNode<T>\* currentPtr = top;

while (currentPtr != nullptr) {

if (currentPtr->data == item)

return &(currentPtr->data);

currentPtr = currentPtr->nextPtr;

}

return nullptr;

}

Очередь – отличие очереди от стека заключается в методах **добавления** и **удаления** элемента, поскольку очередь работает по принципу “первый пришел первый ушел”

template<typename T>

struct QueueNode {

T data;

struct QueueNode\* nextPtr;

};

void enqueue(T value) {

QUEUENODEPTR<T> newPtr = new QUEUENODE<T>;

if (newPtr == nullptr)

return;

newPtr->data = value;

newPtr->nextPtr = nullptr;

if (isEmpty(headPtr))

headPtr = newPtr;

else

tailPtr->nextPtr = newPtr;

tailPtr = newPtr;

}

T dequeue() {

T value = headPtr->data;

QUEUENODEPTR<T> tempPtr = \*headPtr;

headPtr = \*headPtr->nextPtr;

if (headPtr == nullptr)

tailPtr = nullptr;

free(tempPtr);

return value;

}

T peek() {

return headPtr->data;

}

Методы **поиска** и **итерации** аналогичны стеку

template<typename K>

T\* search(K& item) {

QUEUENODEPTR<T> currentPtr = headPtr;

while (currentPtr != nullptr) {

if (currentPtr->data == item)

return &(currentPtr->data);

currentPtr = currentPtr->nextPtr;

}

return nullptr;

}

**Отображение**

void iterate(iteratorCallback<T> callback) {

QUEUENODEPTR<T> currentPtr = headPtr;

while (currentPtr != nullptr) {

callback(&(currentPtr->data));

currentPtr = currentPtr->nextPtr;

}

}

Бинарное дерево – это конечное множество элементов, которое либо пусто, либо содержит элемент (корень), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями. Каждый элемент бинарного дерева называется узлом, а связи между узлами дерева называются его ветвями

template <typename T>

struct TreeNode {

T data;

TreeNode\* left;

TreeNode\* right;

};

При **добавлении**, больший элемент пойдет в правую ветку, а меньший – в левую.

void add(T data) {

root = add(data, root);

}

TreeNode<T>\* add(T data, TreeNode<T>\* node) {

if (node == nullptr) {

node = new TreeNode<T>;

node->data = data;

node->left = nullptr;

node->right = nullptr;

}

else if (data < node->data)

node->left = add(data, node->left);

else node->right = add(data, node->right);

return node;

}

При **удалении** узла, удаляются рекурсивно все ветви:

void deleteNode(TreeNode<T>\* node) {

if (node != nullptr) {

deleteNode(node->left);

deleteNode(node->right);

delete node;

}

}

Inorder

* Сначала посетите все узлы в левом поддереве
* Затем корневой узел
* Посетите все узлы в правом поддереве

void inorder(iteratorCallback<T> callback) {

inorder(root, callback);

}

//обход в симметричном порядке

void inorder(TreeNode<T>\* current, iteratorCallback<T> callback) {

if (current != nullptr) {

inorder(current->left, callback);

callback(&(current->data));

inorder(current->right, callback);

}

}

Preorder

* Посетить все узлы в левом поддереве
* Посетить корневой узел
* Посетить все узлы в правом поддереве

void preorder(iteratorCallback<T> callback) {

preorder(root, callback);

}

//обход в прямом порядке

void preorder(TreeNode<T>\* current, iteratorCallback<T> callback) {

if (current != nullptr) {

callback(&(current->data));

preorder(current->left, callback);

preorder(current->right, callback);

}

}

Postorder

* Посетить все узлы в левом поддереве
* Посетить корневой узел
* Посетить все узлы в правом поддереве

void postorder(iteratorCallback<T> callback) {

postorder(root, callback);

}

//обход в постфиксной форме

void postorder(TreeNode<T>\* current, iteratorCallback<T> callback) {

if (current != nullptr) {

postorder(current->left, callback);

postorder(current->right, callback);

callback(&(current->data));

}

}

**Поиск** элемента по дереву производится благодаря тому, что при добавлении элемента, меньший элемент идет в левую сторону, а больший в правую

template <typename K>

T\* search(K &item) {

return search(root, item);

}

template <typename K>

T\* search(TreeNode<T>\* node, K& item) {

if (node == nullptr)

return nullptr;

if (node->data < item)

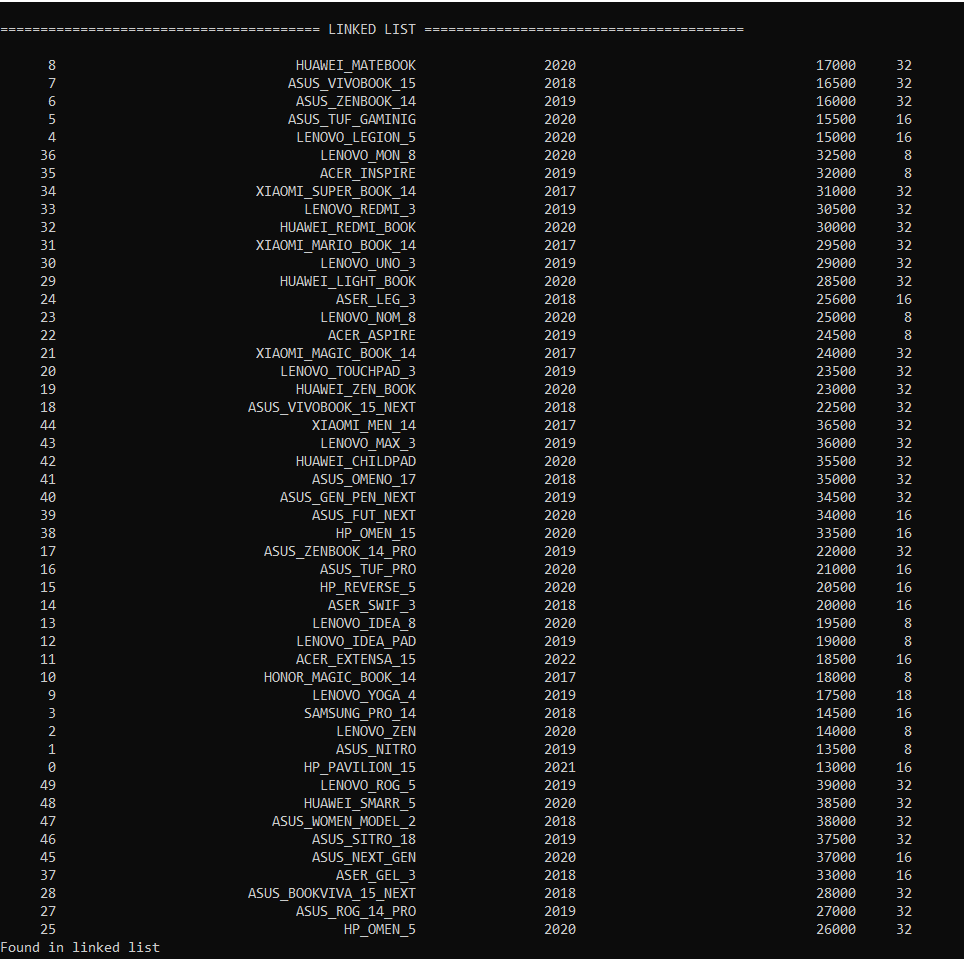
return search(node->right, item);

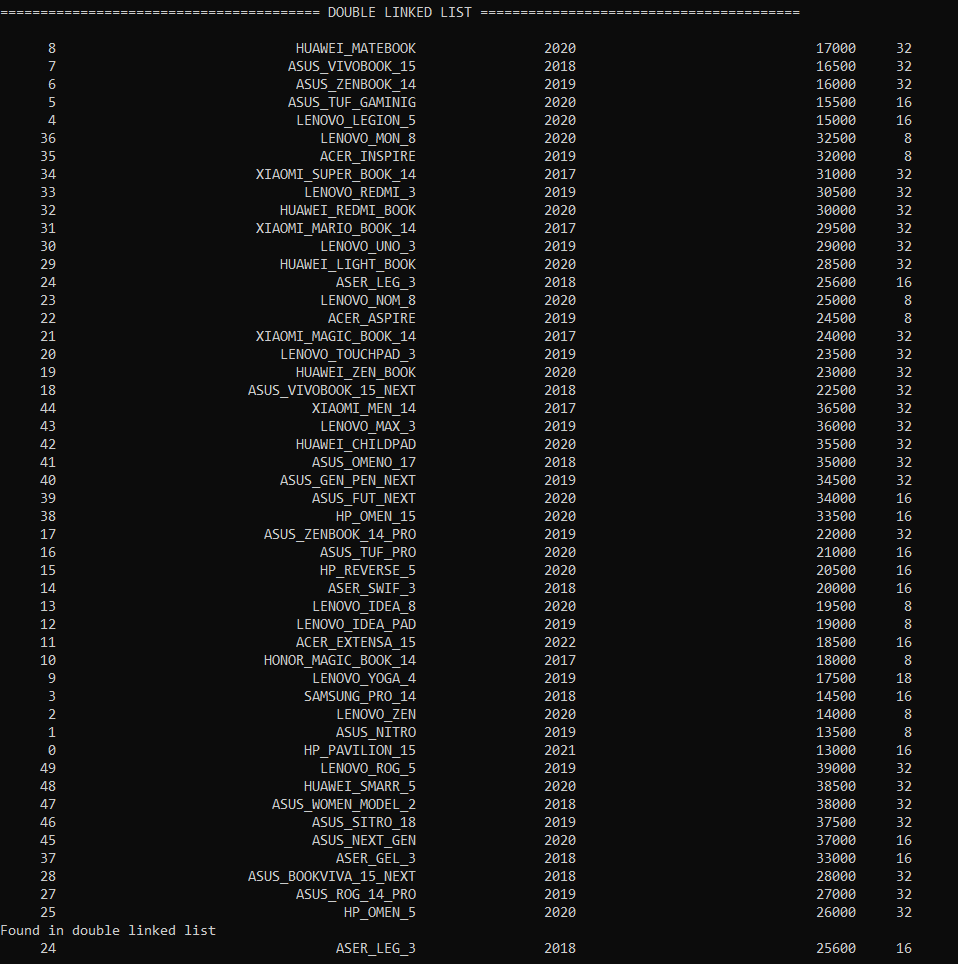
else if (node->data > item)

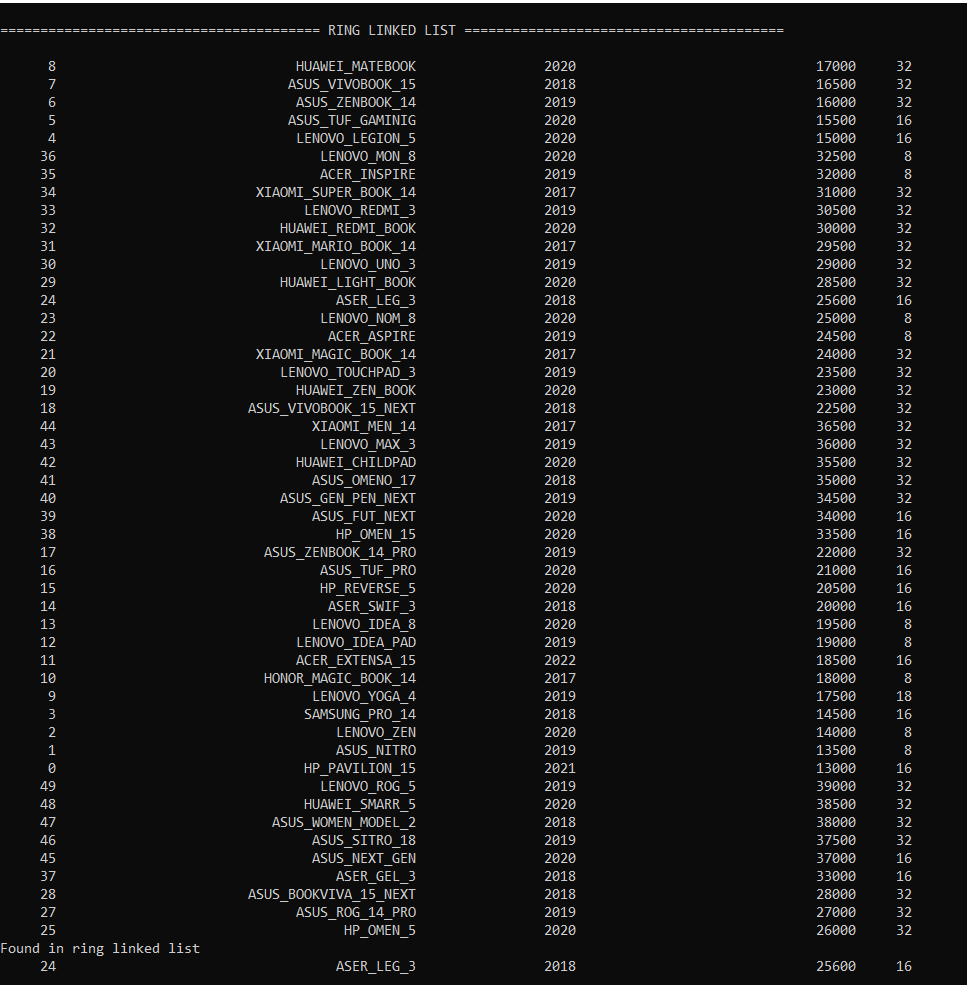
return search(node->left, item);

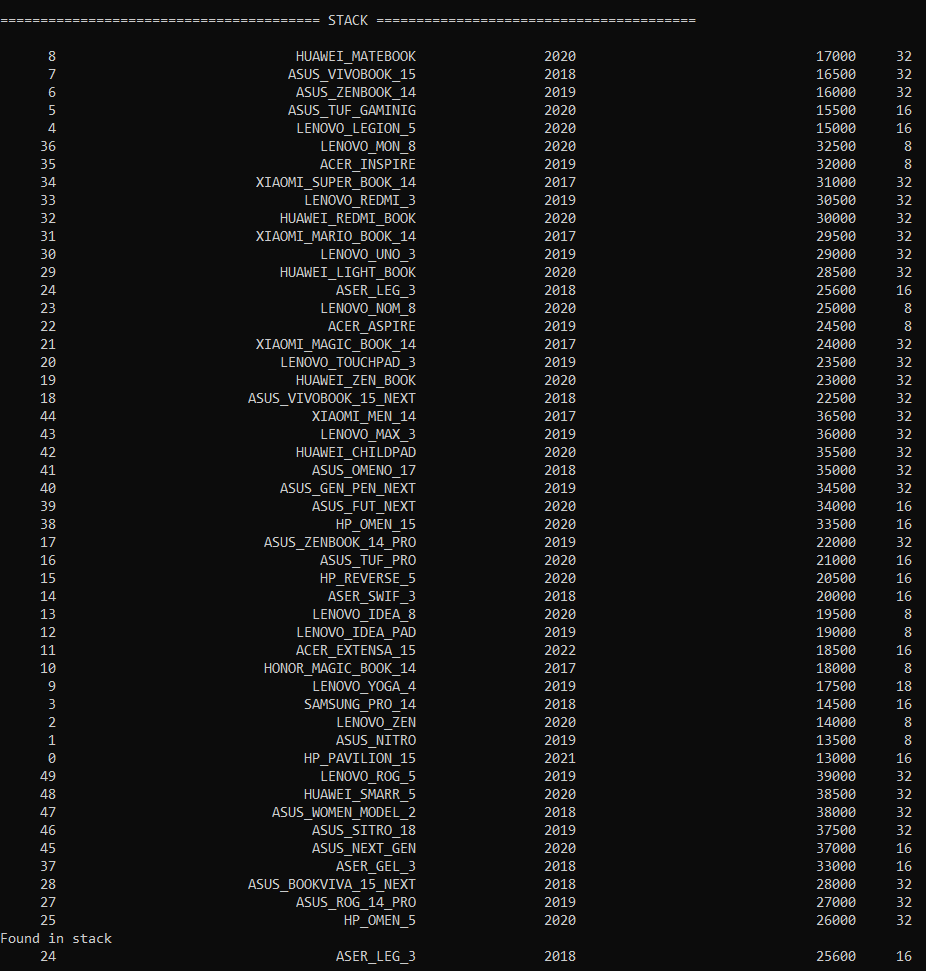
return &(node->data);

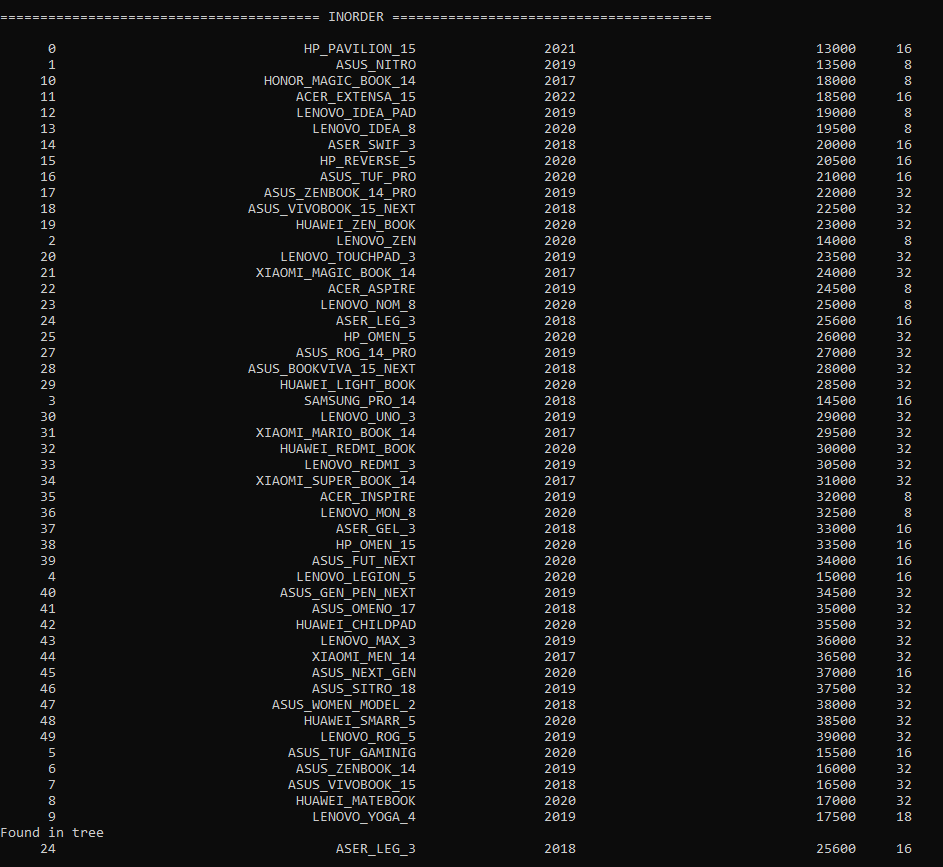
}

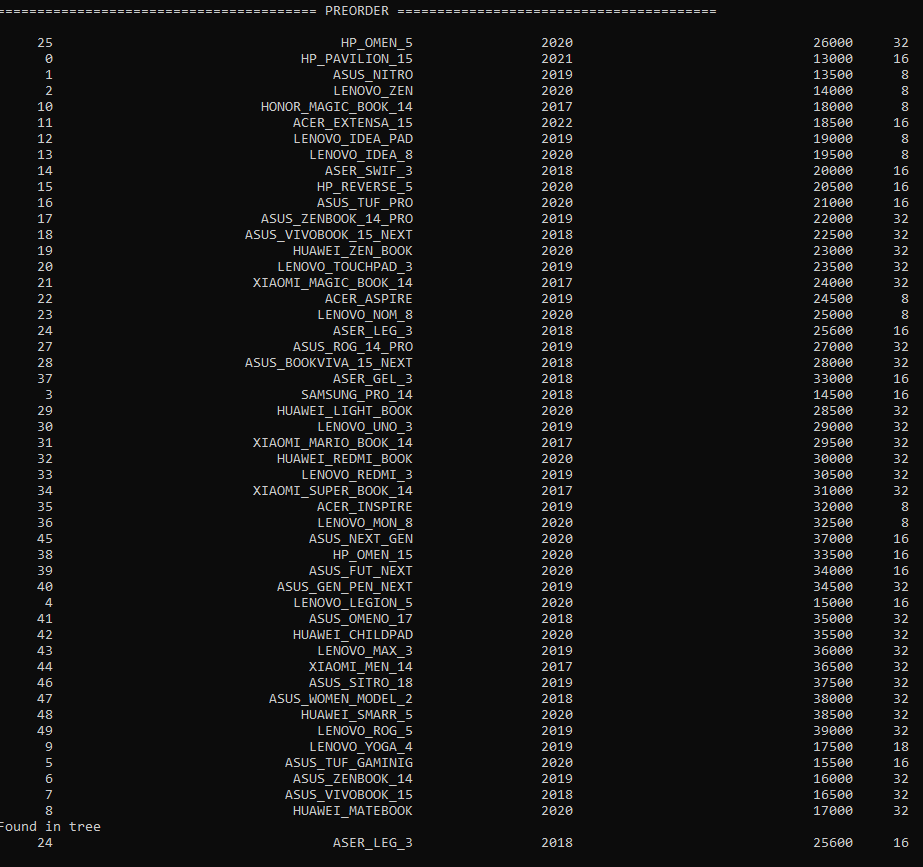


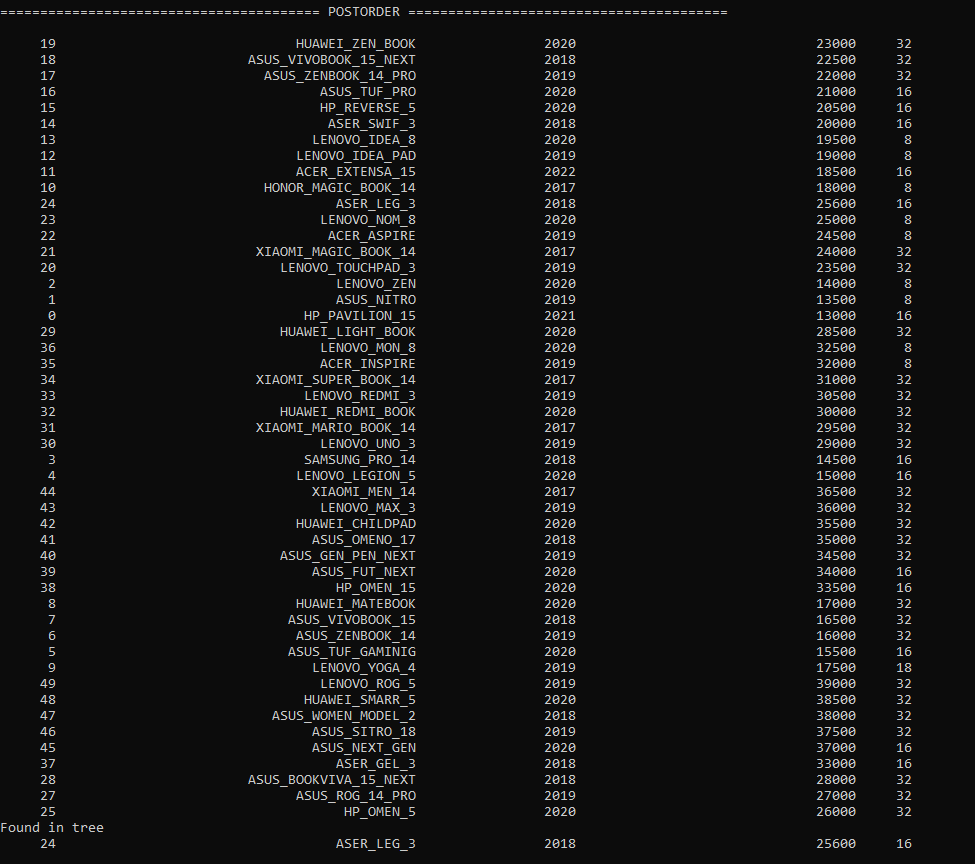














**Выводы**:  
После всей выше проделанной работы можно сделать вывод, что мы усвоили работу с двусвязными, односвязными, кольцевыми списками, а так же с очередями и стеками. В проделанной работе для каждой динамической структуры были использованы и реализованы операции отображения, поиска, вставки, и удаления.