МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Алгоритмы и структуры данных»
Тема: Деревья

Студентка гр. 7381 **Кревчик А.Б.**

Преподаватель **Фирсов М. А.**

Санкт-Петербург
2018

Цель работы.

Ознакомиться с такой структурой данных, как дерево, и научиться применять деревья на практике.

Основные теоретические положения.

Дерево – это конечное множество T, состоящее из одного или более узлов, таких, что

а) имеется один специально обозначенный узел, называемый корнем данного дерева;

б) остальные узлы (исключая корень) содержатся в m >= 0 попарно не пересекающихся множествах T1, T2, ..., Tm, каждое из которых, в свою очередь, является деревом.

Бинарное дерево – дерево, в котором каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками.

Скобочное представление бинарного дерева:

<БД> ::= (< непустое БД >) | <пусто> ,

< непустое БД > ::= (<корень> <БД> <БД>).

Задание.

Вариант 5-д

Для заданного леса с произвольным типом элементов:

а) получить естественное представление леса бинарным деревом;

б) вывести изображение леса и бинарного дерева;

в) перечислить элементы леса в горизонтальном порядке (в ширину).

Описание алгоритма.

Программа осуществляет рекурсивную посимвольную обработку входных данных, игнорируя пробелы и знаки табуляции. Первым элементом, ожидающимся на вход является открывающая круглая скобка ‘(‘, если установлено несоответствие – осуществляется выход из программы с соответствующим сообщением. В успешном случае производится считывание корня некоторого поддерева, после чего рассматривается возможность, что оно закончилось, если это так, то вместе с этим производится проверка на наличие братьев (если текущее поддерево завершено, то согласно скобочной нотации, брат корня этого поддерева лежит “следом” за этим поддеревом). В случае наличия братьев, снова рекурсивно вызывается функция считывания дерева.

Далее, в случае наличия у текущего узла сыновей (т.е. следующим символом входной строки должна быть очередная открывающая скобка), рекурсивно вызывается функция считывания дерева. Последовательно считав сыновей, если таковые имеются, необходимо снова рассмотреть возможность, что дерево закончилось, абсолютно аналогичным способом, как было описано выше.

Очевидно, что, если обрабатываемый символ не оказался ни открывающей, ни закрывающей скобкой, то присутствует ошибка во входной строке.

Способ хранения леса позволяет без каких-либо преобразований вывести соответствующее ему бинарное дерево.

Обход в ширину выполняется стандартным способом с использованием очереди.

Описание задействованных функций и структур данных.

Шаблонный класс ***Tree*** обладает 3-мя свойствами и 1-м методом. Каждый узел леса можно рассматривать как корень некоторого его поддерева, в связи с

чем необходимы сведения о самом узле (***T*** *elem*), первом сыне (***Tree**** *f_son)* и братьях (***Tree**** *next_bro*). Единственный определенный в данном классе метод ***void*** *read_elem()* осуществляет считывание узла (***T*** *elem*).

Функция ***void*** *read_tree****(Tree<T>**** *root****,*** ***Tree<T>**** *bro_pos****,*** ***short*** *lvl****)***осуществляет считывание леса согласно алгоритму, описанному выше, принимая на вход указатель на корень текущего поддерева для которого может найтись брат при текущем или дальнейшем (рекурсивном) считывании, *lvl* – глубину рекурсии.

Функция ***void*** *printBT(Tree<T>* root)* выводит корень бинарного дерева, соответствующего данному лесу.

Функция ***void*** *printSubBT(Tree<T>* root, const std::string& prefix)* рекурсивно выводит бинарное дерево, начиная с правого поддерева и заканчивая левым, добавляя при необходимости в строку *prefix* необходимые отступы, при отсутствии одного из двух узлов-сыновей выводится знак ‘#’.

Функция ***void*** *printSubF(Tree<T>* root, const std::string& prefix)* осуществляет рекурсивный вывод леса, во многом аналогичный уже использованному в функции *printSubBT*, только с учетом того, что количество сыновей не обязано быть меньше либо равным двум.

Функция ***void*** *printF(Tree<T>* root)* аналогично *printBT* выводит корень дерева леса.

Способ хранения дерева в массиве:

Если у текущего узла есть сын, то в поле *f_son* хранится указатель на первого сына. Если у элемента имеются братья, то в поле *next_bro* хранится указатель на следующего за ним брата (слева направо). Если таковые отсутствуют, то поле хранит значение *NULL*. Представление леса (a(b)(c)) (d(e)) в памяти представлено на рис.1.

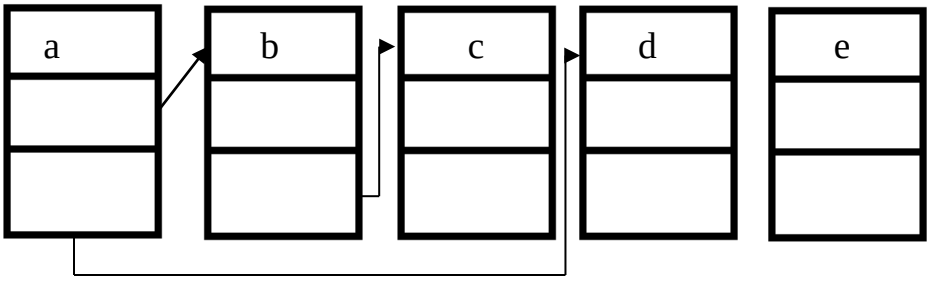

Рисунок 1 – Принцип размещения леса

Результаты тестирования программы представлены в приложении А.

Выводы.

В ходе выполнения работы была изучена новая структура данных – дерево. Получены навыки по работе как с обычным деревом, так и с его бинарным соответствием.

ПРИЛОЖЕНИЕ А
РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ ПРОГРАММЫ

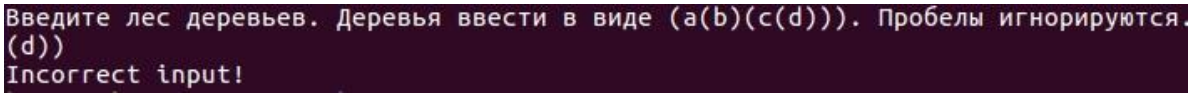

Рисунок 1 – Тест 1

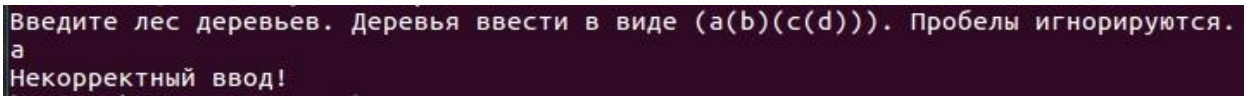

Рисунок 2 – Тест 2

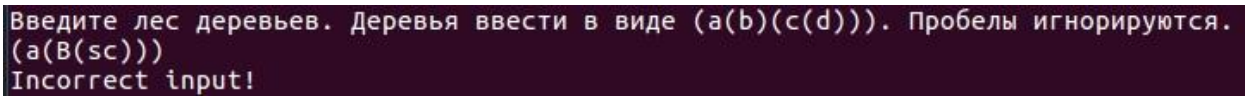

Рисунок 3 – Тест 3

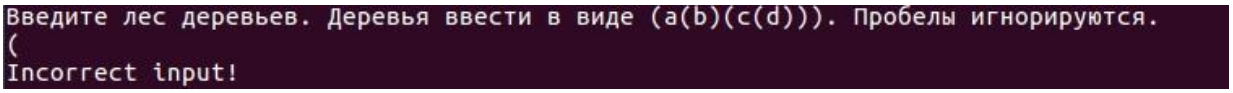

Рисунок 4 – Тест 4

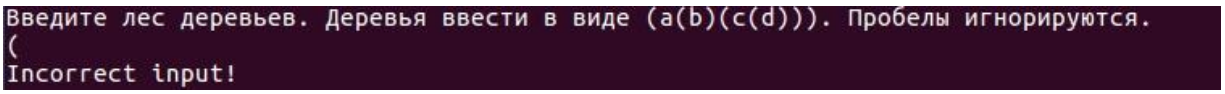

Рисунок 5 – Тест 5

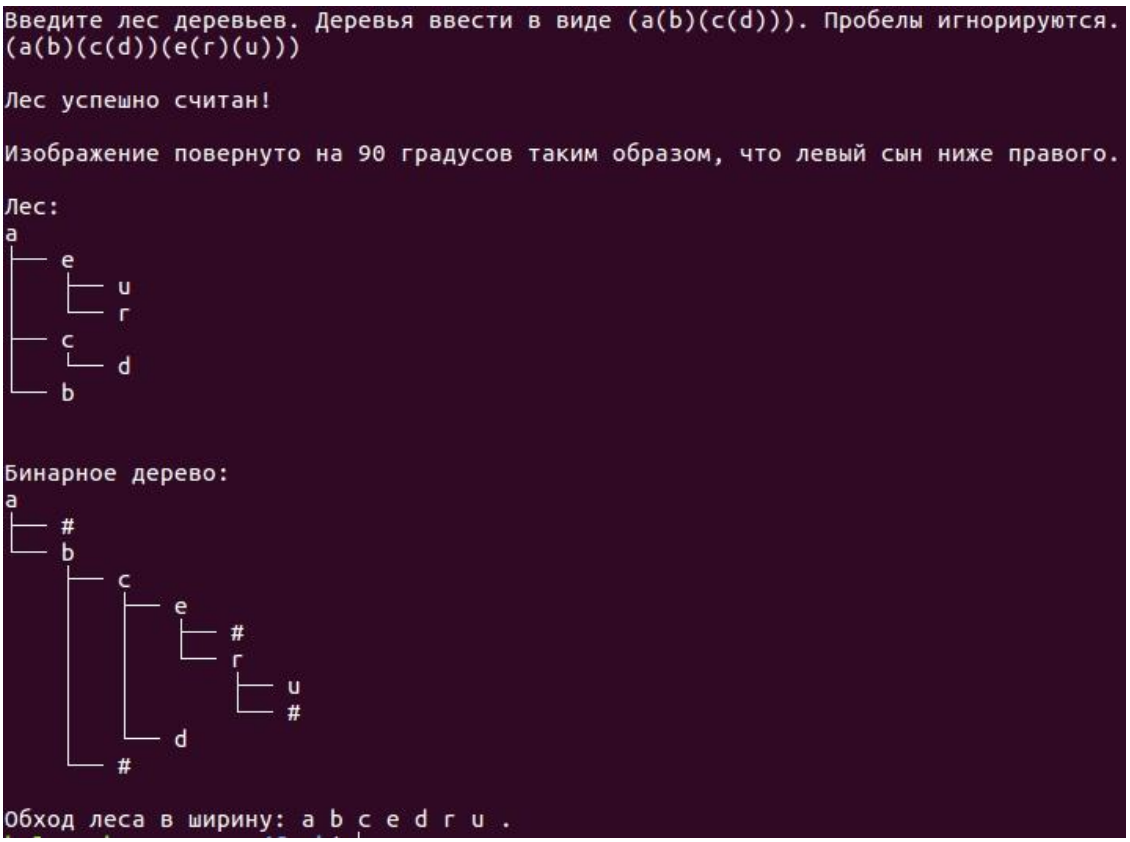

Рисунок 6 – Тест 6

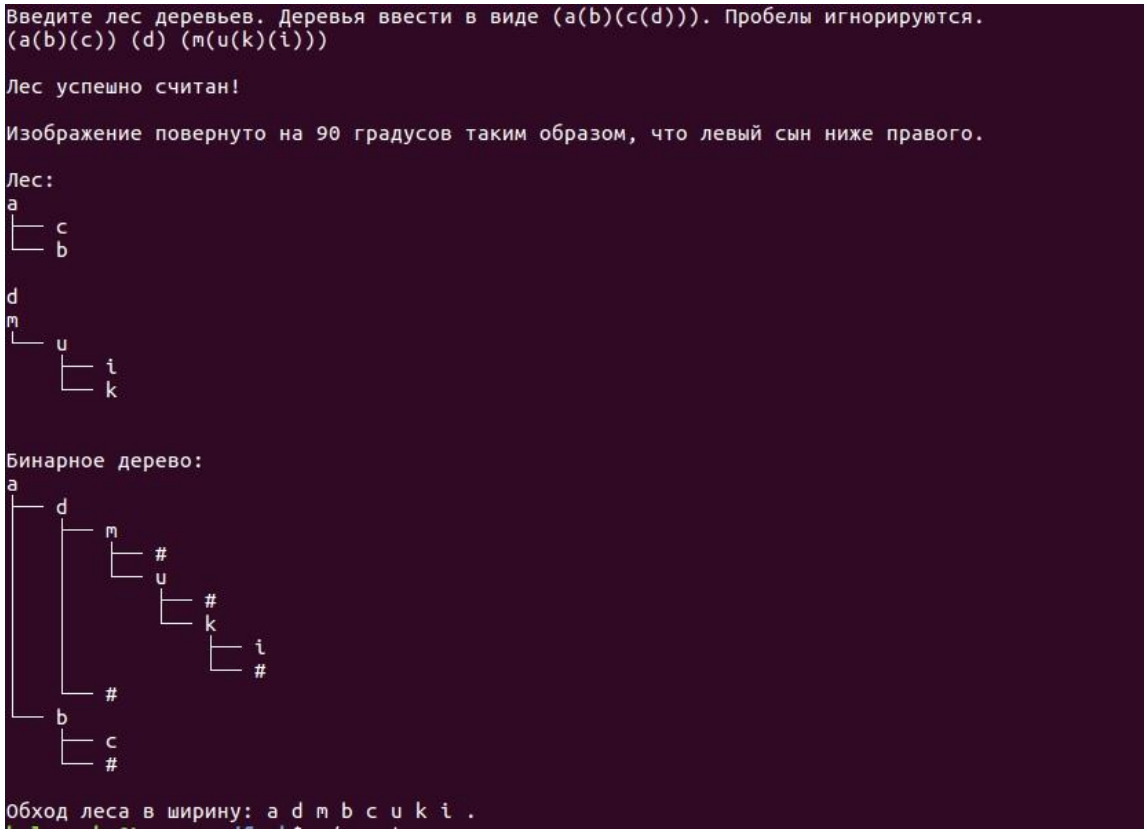

Рисунок 7 – Тест 7

ПРИЛОЖЕНИЕ Б
ГОЛОВНАЯ ФУНКЦИЯ

```
#include <iostream>
#include <string>
#include "t.h"
#include "queue.h"

template <typename T>
void delete_forest(Tree<T>* root){
	if(!root)
		return;
	delete_forest(root->f_son);
	delete_forest(root->next_bro);
	delete root;
	return;
}

int main(){
	Tree<char>* Head = new Tree<char>;
	char c;
	std::cout << "Введите лес деревьев. Деревья ввести в виде 
(a(b)(c(d))). Пробелы игнорируются." << std::endl;
	for(c = getchar(); c == ' ' && c != '\n'; c = getchar());

	if(c == EOF || c == '\n'){
		std::cout << "Некорректный ввод!" << std::endl;
		exit(1);
	}
	while(c != EOF && c != '\n'){
		if(c != '('){
			std::cout << "Некорректный ввод!" << std::endl;
			exit(1);
```

```
            }
            read_tree(Head, Head, 0);
            std::cout << std::endl;
            std::cout << "Лес успешно считан!" << std::endl;
            for(c = getchar(); c == ' ' && c != '\n'; c = getchar());
      }

      std::cout << std::endl;
      std::cout << "Изображение повернуто на 90 градусов таким образом,
что левый сын ниже правого."<< std::endl;
      std::cout << std::endl << "Лес:" << std::endl;
      printF(Head);
      std::cout << std::endl << "Бинарное дерево:" << std::endl;
      printBT(Head);
      std::cout << "Обход леса в ширину: ";

      Queue_Head<Tree<char> *> q; // очередь для хранения указателей на
вершины
    Tree<char>* cur;             // указатель на вершину
    q.push(Head);     // записываем начальные вершины в очередь
    if(Head->next_bro){
      for(cur = Head->next_bro; cur; cur = cur->next_bro)
             q.push(cur);
    }

    while(!q.is_empty()){
        cur = q.give_back();
        std::cout << cur->elem << " ";
        q.del_back();
        if(cur->f_son)
              for(cur = cur->f_son; cur; cur = cur->next_bro)
                 q.push(cur);
```

```
    }
    std::cout<< '.' << std::endl;
      delete_forest(Head);

    return 0;
}
```

ПРИЛОЖЕНИЕ В

ОПИСАНИЕ КЛАССА И МЕТОДОВ, ЗАДЕЙСТВОВАННЫХ ДЛЯ ОБРАБОТКИ ЛЕСА

```
#pragma once

#include <iostream>
#include <stdlib.h>
#include <cstdio>
#include <string>

template <typename T>
class Tree{
public:
	void read_elem();
public:
	T elem;				//элемент
	Tree* f_son;	//указатель на первого сына
	Tree* next_bro; //указатель на первого брата
};

template <typename T>
void Tree<T>::read_elem(){
	char c;
	for(c = getchar(); c == ' ' && c != '\n' && c != EOF; c =
getchar()); //игнорирование пробелов
	if(c == '\n' || c == EOF){
		std::cout << "Incorrect input!" << std::endl;
		exit(1);
	}
	else
		ungetc(c, stdin);
	std::cin >> elem;
	return;
}

template <typename T>
void read_tree(Tree<T>* root, Tree<T>* bro_pos, short lvl){
	char c;
	while(1){
```

```
            root->read_elem();

            for(c = getchar(); c == ' ' && c != '\n' && c != EOF; c =
getchar()); //игнорирование пробелов

            if(c == ')'){
                              //поддерево закончилось
                  root->f_son = NULL;
                  for(c = getchar(); c == ' ' && c != '\n' && c != EOF; c =
getchar());
                  if(lvl){
                        if(c == '('){
                               //есть братья (a(b)(
                              Tree<T>* bro = new Tree<T>;
                              bro_pos->next_bro = bro;
                              read_tree(bro, bro, lvl);
                              return;
                        }
                        if(c == ')'){                    //братьев нет
(a(b))
                              root->next_bro = NULL;
                              ungetc(c, stdin);
                              return;
                        }
                        std::cout << "Incorrect input!" << std::endl;
                        exit(1);
                  }
                  else{  //если нулевой уровень
                        if(c == '('){
                              Tree<T>* bro = new Tree<T>;
                              bro_pos->next_bro = bro;
                              read_tree(bro, bro, lvl);
                              return;
                        }
                        if(c != EOF && c != '\n'){
                              std::cout << "Incorrect input!" << std::endl;
                              exit(1);
                        }
                        else{
                              root->next_bro = NULL;
                              ungetc(c, stdin);
                        }
```

```
                    return;
               }
          }
          if(c == '('){ // есть сыновья
               Tree<T>* son = new Tree<T>;
               root->f_son = son;

                    read_tree(son, son, lvl+1); //опускаемся на
уровень ниже,т.к. нашли сыновей
                    for(c = getchar(); c == ' ' && c != '\n' && c
!= EOF; c = getchar());

          }
          if(c == ')'){ //поддерево закончилось
               for(c = getchar(); c == ' ' && c != '\n' && c != EOF; c =
getchar());
               if(lvl){
                    if(c == '('){ //могут быть братья
                         Tree<T>* bro = new Tree<T>;
                         bro_pos->next_bro = bro;
                         read_tree(bro, bro, lvl);
                         return;
                    }
                    if(c == ')'){
                         root->next_bro = NULL;
                         ungetc(c, stdin);
                         return;
                    }
                    std::cout << "Incorrect input!" << std::endl;
                    exit(1);
               }
               else{
                    if(c == '('){
                         Tree<T>* bro = new Tree<T>;
                         bro_pos->next_bro = bro;
                         read_tree(bro, bro, lvl);
                         return;
                    }
                    if(c != EOF && c != '\n'){
                         std::cout << "Incorrect input!" << std::endl;
                         exit(1);
                    }
```

```
                    else{
                        root->next_bro = NULL;
                        ungetc(c, stdin);
                    }
                    return;
                }
            }
            std::cout << "Incorrect input!" << std::endl;
            exit(1);
        }
}

template <typename T>
void printSubBT(Tree<T>* root, const std::string& prefix){    //вывод БД
    if(!root->f_son && !root->next_bro) //если закончилось
        return;

    std::cout << prefix;
    std::cout <<  "├── ";

    if(root->next_bro){                          //правый сын

        std::string newPrefix = prefix + "│   ";
        std::cout << root->next_bro->elem << std::endl;
        printSubBT(root->next_bro, newPrefix);
    }
    else
     std::cout << "#" << std::endl;

    if(root->f_son){
        std::cout << prefix << "└── " << root->f_son->elem << std::endl;
     //левый сын
        printSubBT(root->f_son, prefix + "    ");
    }
    else
     std::cout << prefix << "└── #" << std::endl;
}

template <typename T>
void printBT(Tree<T>* root){ // вывод корня БД
    std::cout << root->elem << std::endl;
    printSubBT(root, "");
```

```
    std::cout << std::endl;
}

template <typename T>
void printSubF(Tree<T>* root, const std::string& prefix){  //вывод леса
    Tree<T>* printed_bro = root;
    while(1){
     Tree<T>* tmp;
     for(tmp = root; tmp->next_bro; tmp = tmp->next_bro){
          if(printed_bro && tmp->next_bro == printed_bro)
               break;
     }
     std::cout << prefix;
     if(root != tmp){
          std::cout <<  "├── ";
          std::string newPrefix = prefix + "│   ";
           std::cout << tmp->elem << std::endl;
           if(tmp->f_son)
               printSubF(tmp->f_son, newPrefix);
        }
        else{
           std::cout <<  "└── " << root->elem << std::endl;
           if(root->f_son){
                printSubF(root->f_son, prefix + "    ");
           }
           return;
        }
        printed_bro = tmp;
    }
}

template <typename T>
void printF(Tree<T>* root){                //вывод корня дерева леса
     std::cout << root->elem << std::endl;
     if(root->f_son){
          printSubF(root->f_son, "");
          std::cout << std::endl;
     }
     Tree<T>* tmp;   //след дерево леса
     for(tmp = root->next_bro; tmp; tmp = tmp->next_bro){//след дерево
леса
          std::cout << tmp->elem << std::endl;
```

```
            if(tmp->f_son){
                  printSubF(tmp->f_son, "");
                  std::cout << std::endl;
            }
      }
}
```

ПРИЛОЖЕНИЕ Г

ОПИСАНИЕ КЛАССА И МЕТОДОВ ОЧЕРЕДИ

```
#pragma once  // file include only one time
#include "queue.h"
template <class Type> // class element of queue
class Queue{
public:
    Queue();
    ~Queue();
public:
    class Queue* next;
    Type data;
};

template <class Type> // class head of queue
class Queue_Head{
public:
    Queue_Head();
    ~Queue_Head();
    void del_back();
    Type give_back();
    void push(Type);
    void print_data();
    bool is_empty();
private:
```

```
    class Queue<Type>* head;
};

template <class Type>
Queue_Head<Type>::Queue_Head(){
    head = new Queue<Type>;
    head->next = NULL;
}

template <class Type>
Queue_Head<Type>::~Queue_Head(){ // free memory to head
    delete head;
}

template <class Type>
Queue<Type>::Queue(){
    next = NULL;
}

template <class Type>
Queue<Type>::~Queue(){  // free memory to elements of queue
    delete this->next;
}

template <class Type>
```

```
void Queue_Head<Type>::push(Type data){ // add element at the begin of
queue
    Queue<Type>* temp = new Queue<Type>;
    temp->next = head->next;
    head->next = temp;
    temp->data = data;
}


template <class Type>
Type Queue_Head<Type>::give_back(){ // give last element of queue, but
don`t delete it
    Queue<Type>* temp;
    // if(!head->next)
    //     return NULL;
    temp = head->next;
    if(!temp->next)
        return head->next->data;
    while(temp->next)
        temp = temp->next;
    return temp->data;
}


template <class Type>
void Queue_Head<Type>::del_back(){ // delete last element of queue
     Queue<Type>* temp;
    if(!head->next)
        return;
```

```
    temp = head->next;
    if(!temp->next){
        delete head->next;
        head->next = NULL;
        return;
    }
    while(temp->next->next)
        temp = temp->next;
    delete temp->next;
    temp->next = NULL;
}

template <class Type>
bool Queue_Head<Type>::is_empty(){ // checked to empty queue
    if(!head->next)
        return true;
    return false;
}

template <class Type>
void Queue_Head<Type>::print_data(){ // print all queue
    Queue<Type>* temp = new Queue<Type>;
    temp = head->next;
    if(!head->next)
        std::cout << "queue is empty!" << std::endl;
    while(temp){
```

```
            std::cout << temp->data;
            temp = temp->next;
        }
        delete temp;
    }
```