

WAYS TO LEARN AND FACILITATE THE DIFFICULTIES IN THE PROCESS OF WORKING WITH LISTS ON THE DATA STRUCTURE AND ALGORITHMS MODULE

NURJABOVA DILAFRUZ SHUKRULLAEVNA

Applicant, Researcher Tuit Karshi branch, department Software Engineering, Karshi,
Uzbekistan. dilyaranur1986@gmail.com

ULASHEVA SHAXLO TAGAEVNA,

Student Tuit Karshi branch, department Software Engineering, Karshi, Uzbekistan

RUSTAMOV MAKRUUF AKMALOVICH.

Student Tuit Karshi branch, department Software Engineering, Karshi, Uzbekistan

ABSTRACT:

This article is written about the relief of working with programs in the educational process, that is, in order to increase the knowledge of the students and learners. Using programming, it is important to use the data structure and their algorithms, implementing it, obtaining results in practice. The use of lists in the Dynamic Data Structure in different views is also mentioned separately.

KEYWORDS: List, Dynamic Data Structure, algorithm, module, node, link, memory, pointer.

INTRODUCTION;

Often in serious programs, you need to use data whose size and structure must change as you work. Dynamic arrays do not help here, because you cannot say in advance how much memory you need to allocate – this is only found out in the process of working. For example, you need to analyze the text and determine which words and how many occur in it, and these words need to be arranged alphabetically.

In such cases, use data of a special structure, which are separate elements associated with links.

Each element (node) consists of two

memory areas: data fields and links. Links are addresses of other nodes of the same type that this element is logically associated with. In the C language for the organization of the links used variables are pointers. When adding a new node to such a structure, a new block of memory is allocated and (using links) links are established between this element and existing ones. Null references are used to indicate the final element in the chain.

In the simplest case, each node contains only one link. For certainty, we assume that the problem of frequency analysis of the text is solved-determining all the words that occur in the text and their number. In this case, the item's data area includes a string (no more than 40 characters long) and an integer.

Each element also contains a link to the next element. The last item in the list has a null reference field. In order not to lose the list, we must store the address of its first node somewhere (in a variable) – it is called the "head" of the list. In the program, you need to declare two new data types-the node of the Node list and the PNode pointer to it. A node is a structure that contains three fields-a string, an integer, and a pointer to the same node. Rules of the C language allowed the ad

In the future, we will assume that the head pointer points to the beginning of the list,

that is, declared as

```
PNode Head = NULL;
```

The first letter "P" in the PNode type name comes from the word pointer.) At the beginning of the work, there is no element in the list, so the null address is written to the head pointer.

In order to add a node to the list, you must create it, that is, allocate memory for the node and remember the address of the selected block. We will assume that we need to add a node to the list that corresponds to the new word that is written in the new word variable. Let's make a function that creates a new node in memory and returns its address. Note that when writing data to a node, the structure fields are accessed via a pointer.

After that, the node must be added to the list (at the beginning, at the end, or in the middle).

When adding a new NewNode node to the top of the list, 1) set the newNode node reference to the head of the existing list, and 2) set the head of the list to the new node.

This is how the AddFirst procedure works. It is assumed that the address of the beginning of the list is stored in Head. It is important that the address at the beginning of the list is passed by reference here and later, because when a new node is added, it changes within the procedure.

Given the NewNode address of the new node and the p address of one of the existing nodes in the list. You need to insert a new node after the node with address p in the list. This operation is performed in two stages:

- 1) set the link of the NewNode to the node following the data;
- 2) Set the link of this node p to NewNode.

You can't change the sequence of operations, because if you change the link at node p first, the address of the next node will be lost.

This addition scheme is the most complex. The problem is that in the simplest linear list (it is called a single-linked list, because the links are directed only in one direction), in order to get the address of the previous node, you need to go through the entire list first.

The task is reduced to either inserting a node at the beginning of the list (if the specified node is the first one), or inserting it after the specified node.

This procedure provides "foolproof" protection: if a node is specified that is not present in the list, then at the end of the loop, the q pointer is NULL and nothing happens.

There is another interesting trick: if you need to insert a new node NewNode to the set of node p insert a node after this node, and then performs data exchange between nodes p and NewNode. Thus, at p in fact, there will be a node with new data, and the address of NewNode – with the data that was in the p site, i.e. we solved the problem. This technique will not work if the address of the new node NewNode is stored somewhere in the program and then used, because this address will contain other data. Solving the problem, you must first find the last node whose reference is NULL, and then use the insertion procedure after the specified node. Separately, you need to handle the case when the list is empty.

In order to go through the entire list and do something with each of its elements, you need to start with head and use the next pointer to move to the next node.

Often you need to find the desired item in the list (its address or data). Note that the required item may not exist, so the view ends when the end of the list is reached. This approach leads to the following algorithm:

- 1) Start with the head of the list;
- 2) While the current element exists (the pointer is not NULL), check the required condition and go to the next element;

3) Finish when the required item is found or all items in the list are viewed.

For example, the following function searches the list for an item that matches the specified word (for which the word field matches the specified New Word string) and returns its address or NULL if there is no such node.

Let's return to the problem of constructing an alphabetic-frequency dictionary. In order to add a new word to the desired location (in alphabetical order), you need to find the address of the node before which you want to insert a new word. This will be the first node from the beginning of the list for which the "its" word will be "more" than the new word. Therefore, it is enough to simply change the condition in the while loop in the Find function. Given that the strcmp function returns the "difference" of the first and second words.

This function returns the address of the node before which the new word should be inserted (when the strcmp function returns a positive value), or NULL if the word should be added to the end of the list.

Now you can fully write a program that processes the input file.txt and compiles an alphabetic-frequency dictionary for it in the output file.txt.

The variable n stores the value returned by the fscanf function (the number of successfully read elements). If this number is less than one (the read failed or the data in the file ran out), the while loop exits.

First, we try to search for this word in the list using the Find function. If found simply increase the counter of the found node. If the word is encountered for the first time, a new node is created in memory and filled with data. Then use the Find Place function to determine which node in the list to add it to.

When the list is ready, open the file for output and, using a standard pass through the

list, output the found words and counter values.

This procedure is also associated with searching for a given node throughout the list, since we need to change the link from the previous node, and it is not possible to go directly to it. If we find a node that is followed by the node being deleted, we just need to rearrange the link.

The case when the first item in the list is deleted is handled separately. Deleting a node frees up the memory it used to occupy.

Separately, we consider the case when the first item in the list is deleted. In this case, the address of the node being deleted is the same as the address of the head of the Head list, and you just need to write the address of the next element to the Head.

You have noticed that for the list variant considered, you need to handle border cases separately: adding to the beginning, adding to the end, and deleting one of the extreme elements. You can greatly simplify the above procedures if you set two barriers-the dummy first and last elements. Thus, there are always at least two barrier elements in the list, and all working nodes are located between them.

Many problems when working with a single-linked list are caused by the fact that it is impossible to go to the previous element in them. It is a natural idea to store a link in memory not only to the next item, but also to the previous item in the list. To access the list, not one pointer variable is used, but two-a reference to the "head" of the list (Head) and to the "tail" - the last element (Tail). Each node contains (in addition to useful data) a link to the next node (the next field) and the previous one (the prev field). The next field for the last element and the prev field for the first element contain NULL. The node is declared as follows:

In the future, we will assume that the head pointer points to the beginning of the list,

and the Tail pointer points to the end of the list:

For an empty list, both pointers are NULL.

Operations with a two-linked list

➤ Adding a node to the top of the list

When adding a NewNode to the top of the list, you should

1) Set the newNode node's next link to the head of an existing list and its prev link to NULL;

2) set the prev link of the former first node (if it existed) to NewNode;

3) set the list head to a new node;

4) If there were no items in the list, the tail of the list is also set to the new node.

The following procedure works according to this scheme:

Due to the symmetry, adding a NewNode to the end of the list is quite similar. In the procedure, you need to replace Head with Tail and Vice versa, as well as change prev and next.

Given the NewNode address of the new node and the p address of one of the existing nodes in the list. You need to insert a new node after p in the list. If p is the last node, the operation is reduced to adding it to the end of the list (see above). If node p is not the last node, the insertion operation is performed in two stages:

1) Set the new node's links to the next node after the data (next) and the one before it (prev);

2) set the links of neighboring nodes to include NewNode in the list.

This method is implemented by the following procedure (it also takes into account the possibility of inserting an element at the end of the list, which is why the parameters are passed references to the head and tail of the list):

Adding a node before the specified one is done the same way.

Passing through a two-linked list can be

performed in two directions – from head to tail (as for a single-linked list) or from tail to head.

This procedure also requires a reference to the head and tail of the list, because they may change when you delete the last item in the list. The first step is to set the links of neighboring nodes (if there are any) as if the node being deleted would not exist. Then the node is deleted and the memory it occupies is freed. These steps are shown in the figure below. Separately, it checks whether the node being deleted is the first or last node in the list.

Sometimes a list (single-linked or double-linked) is closed in a ring, that is, the next pointer of the last element points to the first element, and (for double-linked lists) the prev pointer of the first element points to the last. In such lists, the concept of the "tail" of the list does not make sense to work with it, you need to use a pointer to the "head", and any element can be considered a "head".

We can develop programs in a convenient way using the above lists. For example, in the following program, such conditions as adding an item to the list in such a way that the list is bewrilgan and the queue is used, deleting the item, removing the above element, knowing the length of the queue are included.

Int data [N]; the elements of the list of the whole type are given in the form of a mass, which is equal to N. Structure Queue we will declare the queue in structure view. The beginning of the list is Int first; if Int last; end of the list. In Uhbu program we will be announcing the list in turn view as well as its elements in a massive view. Void Creation (Queue *Q) function the name of the queue that does not return a value. {Q ->first=Q ->last=1 ;}-Determine the queue through Q and we know that there is an indicator at the beginning and end of the list and equalize them. The queue from this we know that the last element is equal to 1. Bool Full (Queue *Q) –we check if

the list items are empty or not empty.

If (Q->last==Q->first) return true - true if the condition is fulfilled, otherwise false Else return false; Void Add(Queue *Q)-add elements of the function's value non-returnable queue; Int value-whole value; If ((Q->last%(N-1))+1==Q->first) if the sum together of the last queue measurement of less than one is equal to its first element, otherwise Q->data[Q->last]=value-to the value,q->last=(Q->last%(n-1))+1-If equal to the last element, respectively, the element is added; void delete(queue *Q)-the element of the queue that does not return the value is deleted; q->First = (Q->first%(N-1))+1-if the first index is equal to the percentage of the first low element, the queue element is deleted; int;

{Return Q ->data [Q->first];} - sets the value; Int Size (Queue *Q)-queue width;

If (Q->first>Q->last) if the first item is on the floor from the last item, return (N-1)-(Q->first-Q->last)-returns a value, otherwise Else return Q->last-Q->first; Void main () - main function; Setlocale(LC_ALL,"Russian")-for text application in Russian; Queue Q-call queue; Creation(&Q) - create queue;

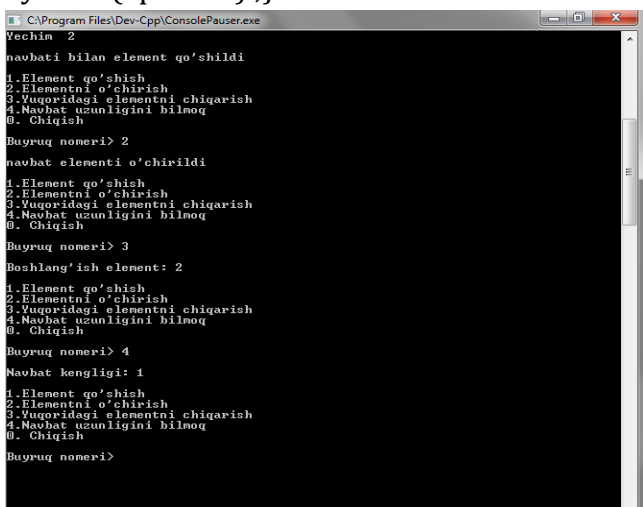
```
#include <iostream>
Using namespace STD;
Const int N=6; // turn scale
Struct Queue
{
Int data [N]; // Massiv's information
Int first; // initial indicator
Int last; // latest indicator
};
Void Creation (Queue *Q) // create queue
{Q->first=Q->last=1 ;}
Bool Full (Queue *Q) // check the gap of the queue
{
If (Q->last==Q->first) return true;
Else return false;
}
```

```
Void Add (Queue *Q) // add element
{
Int value;
Cout<<>\nYechim > «; cin>>value;
If ((Q->last % ( N-1)) +1==Q->first)
Cout<<>\n..... \n\n»;
Else
{
Q->data [Q->last] =value;
Q->last= (Q->last % ( N-1)) +1;
Cout<<endl>> added item respectively \n\n»;}}
Void Delete (Queue *Q) // item deleted
{
Q->first= (Q->first % ( N-1)) +1;
Cout<<endl>> the queue item is deleted \n\n»;
}
Int Top (Queue *Q) // extraction of the starting element

{Return Q->data [Q->first];}
Int Size (Queue *Q) // turn width
{
If (Q->first>Q->last) return (N-1)-(Q->first-Q->last);
Else return Q->last-Q->first ;}
Void main () // main function
{
Setlocale(LC_ALL,»Rus»);
Queue Q;
Creation (&Q);
Char number;
Do {
Cout<<>>1. Add item » <<endl;
Cout<<>>2. Delete item »<<endl;
Cout<<>>3. Removing the above item »<<endl;
Cout<<>>4. Knowing the length of the queue
»<<endl;
Cout<<>>0. Output \n\n»;
Cout<<>> Command number > «; cin>>number;
Switch (numbe) {
Case '1': Add (&Q);
Break;
//-----
Case '2':
```

```
If (Full (&Q)) cout<<endl<<» The queue is empty
\n\n»;
Else Delete (&Q);
Break;
//-----
Case '3':
If (Full (&Q)) cout<<endl<<» The queue is empty
\n\n»;
Else cout<<»\n item: <<<Top (&Q) <<<\n\n»;
Break;
//-----
Case '4':
If (Full (&Q)) cout<<endl<<» The queue is empty
\n\n»;
Else cout<<»\n item width: <<<Size (&Q) <<<\n\n»;
Break;
//-----
Case '0': break;
Default: cout<<endl<<» command is clear \n\n»;
Break ;}} while (number! ='0');
System («pause» );}
```

- 2) Resolution of the Cabinet of Ministers of the Republic of Uzbekistan "On measures to further improve the Governmental portal of the Republic of Uzbekistan on the Internet with regard to the provision of open data" of August 7, 2015 №. 232.
- 3) J. A. Day and J. Web lecture intervention in a human-computer interaction course," IEEE Transactions on Education, vol. 49, no. 4, pp. 420–431, November 2006.



```
C:\Program Files\Dev-Cpp\ConsolePauser.exe
Vechin 2
navhati bilan element qo'shildi
1. Element qo'shish
2. Elementni o'chirish
3. Yuqoridagi elementni chiqarish
4. Navbat uzunligini bilmoq
5. Chiqish
Buyruq noneri> 2
navbat elementi o'chirildi
1. Element qo'shish
2. Elementni o'chirish
3. Yuqoridagi elementni chiqarish
4. Navbat uzunligini bilmoq
5. Chiqish
Buyruq noneri> 3
Boshlang'ish element: 2
1. Element qo'shish
2. Elementni o'chirish
3. Yuqoridagi elementni chiqarish
4. Navbat uzunligini bilmoq
5. Chiqish
Buyruq noneri> 4
Navbat kengligi: 1
1. Element qo'shish
2. Elementni o'chirish
3. Yuqoridagi elementni chiqarish
4. Navbat uzunligini bilmoq
5. Chiqish
Buyruq noneri>
```

In conclusion, these amenities will give their effect in the future. A number of works being done in our country are developing in vain. In addition to this work, we should work together with young people.

REFERENCES:

- 1) Adam Drozdek. Data structure and algorithms in C++. Fourth edition. 2013. Chapter 3.