Динамические структуры данных

Прикладное программирование

Стеки

В списках доступ к элементам происходит посредством адресации, при этом доступ к отдельным элементам не ограничен. Но существуют также и такие списковые структуры данных, в которых имеются ограничения доступа к элементам. Одним из представителей таких списковых структур является стековый список или просто стек.

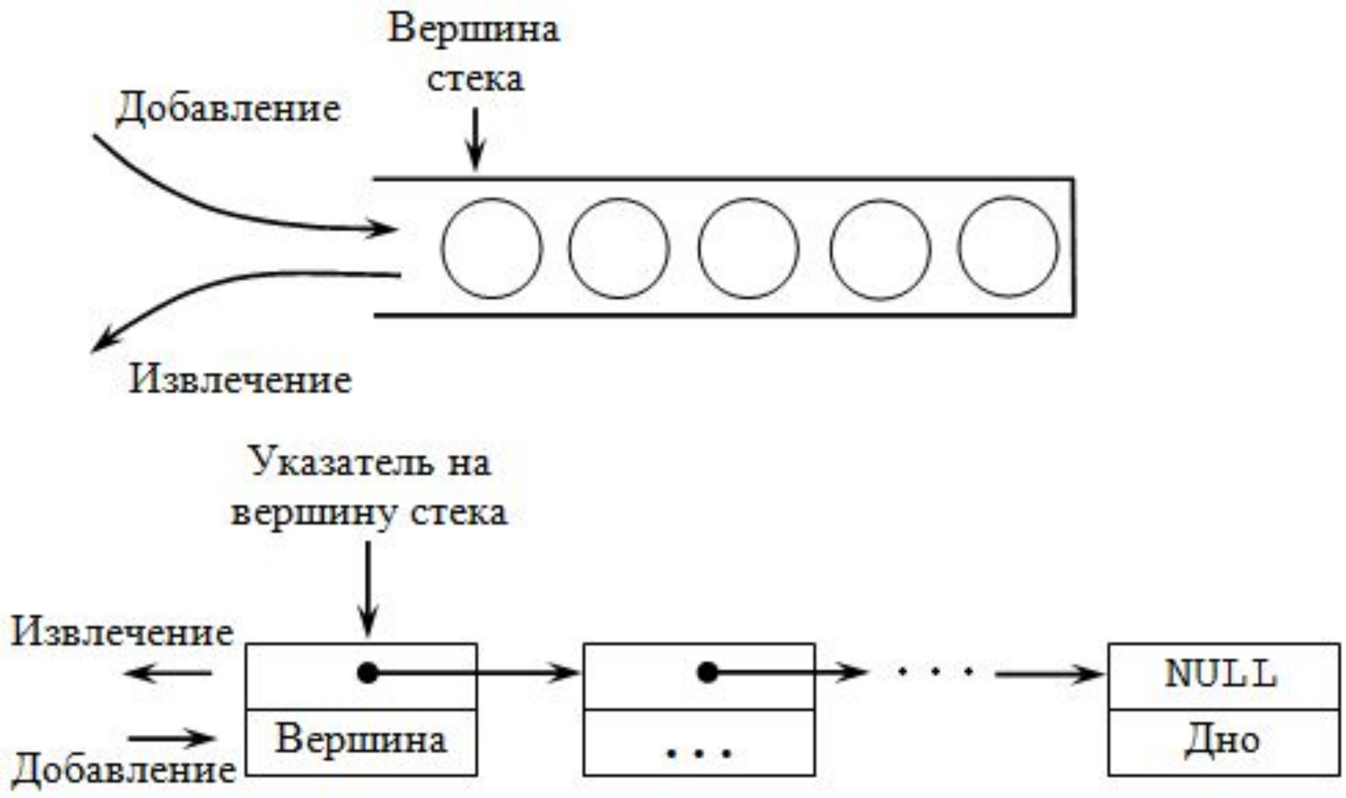
Стеки

- Стек (англ. stack – стопка) – это структура данных, в которой новый элемент всегда записывается в ее начало (вершину) и очередной читаемый элемент также всегда выбирается из ее начала. В стеках используется метод доступа к элементам LIFO (Last Input – First Output, "последним пришел – первым вышел"). Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно сначала взять верхнюю.

Стеки

- Стек – это список, у которого доступен один элемент (одна позиция). Этот элемент называется вершиной стека. Взять элемент можно только из вершины стека, добавить элемент можно только в вершину стека. Например, если записаны в стек числа 1, 2, 3, то при последующем извлечении получим 3,2,1.

Стеки


Вершина стека
Добавление
Извлечение
Указатель на вершину стека
Извлечение
Добавление
Вершина
...
NULL
Дно

Описание стека

- Описание стека выглядит следующим образом:
- struct имя_типа {
- информационное поле;
- адресное поле;
- };

Описание стека

- где информационное поле – это поле любого ранее объявленного или стандартного типа;
- адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента стека.

Описание стека

- Например:
- struct list {
-             type pole1;
-             list *pole2;
-           } stack;

Организация стека

- Стек как динамическую структуру данных легко организовать на основе линейного списка. Поскольку работа всегда идет с заголовком стека, то есть не требуется осуществлять просмотр элементов, удаление и вставку элементов в середину или конец списка, то достаточно использовать экономичный по памяти линейный однонаправленный список.

Организация стека

- Для такого списка достаточно хранить указатель вершины стека, который указывает на первый элемент списка. Если стек пуст, то списка не существует, и указатель принимает значение NULL.

Описание стека

- **Описание элементов стека аналогично описанию элементов линейного однонаправленного списка. Поэтому объявим стек через объявление линейного однонаправленного списка:**
- `struct Stack {`
- `        Single_List *Top;//вершина стека`
- `      };`
- `……….`
- `Stack *Top_Stack;//указатель на вершину стека`

Основные операции, производимые со стеком

- Основные операции, производимые со стеком:

✓ создание стека;

✓ печать (просмотр) стека;

✓ добавление элемента в вершину стека;

✓ извлечение элемента из вершины стека;

✓ проверка пустоты стека;

✓ очистка стека.

Создание стека

- в функции создания стека используется функция добавления элемента в вершину стека.
- //создание стека
- void Make_Stack(int n, Stack* Top_Stack){
-   if (n > 0) {
-     int tmp;//вспомогательная переменная
-

Создание стека

- cout << "Введите значение ";
- cin >> tmp; //вводим значение информационного поля
- Push_Stack(tmp, Top_Stack);
- Make_Stack(n-1,Top_Stack);
- }
- }

Добавление элемента в вершину стека

- //добавление элемента в вершину стека
- void Push_Stack(int NewElem, Stack* Top_Stack){
- Top_Stack->Top =Insert_Item_Single_List(Top_Stack->Top,1,NewElem);
- }

Печать стека

```
//печать стека
void Print_Stack(Stack* Top_Stack){
  Print_Single_List(Top_Stack->Top);
}
```

Извлечение элемента из вершины стека

- //извлечение элемента из вершины стека
- int Pop_Stack(Stack* Top_Stack){
- int NewElem = NULL;
- if (Top_Stack->Top != NULL) {
- NewElem = Top_Stack->Top->Data;

Извлечение элемента из вершины стека

- Top_Stack->Top = Delete_Item_Single_List(Top_Stack->Top,0);
- //удаляем вершину
- }
- return NewElem;
- }

Проверка пустоты стека

- //проверка пустоты стека
- bool Empty_Stack(Stack* Top_Stack){
-   return Empty_Single_List(Top_Stack->Top);
- }

Очистка стека

- //очистка стека
- void Clear_Stack(Stack* Top_Stack){
- Delete_Single_List(Top_Stack->Top);
- }

Пример работы со стеком

- Пример. Дана строка символов. Проверьте правильность расстановки в ней круглых скобок.

- В решении данной задачи будем использовать стек. Приведем главную функцию и функцию для проверки правильности расстановки круглых скобок.

Пример работы со стеком

- //главная функция
- int main()
- {
- char text[255];
- printf("Введите текст, содержащий \"(\" и \")\" \n");
-

Пример работы со стеком

- gets(text);
- Check_Brackets (text);
- system("pause");
- return 0;
- }

Пример работы со стеком

//функция проверки правильности расстановки скобок

```
void Check_Brackets (char *text){
  int i;
  int flag=1;
  Stack *Top_Stack;
  Top_Stack = new Stack();
```

Пример работы со стеком

```
for(i=0; i<strlen(text); i++) {
  if(text[i]==')' ) {
    if(Empty_Stack(Top_Stack)) {
    //Попытка удалить нулевой элемент стека
      flag=0;
      break;
    }
```

Пример работы со стеком

```
if(Top_Stack->Top->Data == '(')
        Pop_Stack(Top_Stack);
    else {
        flag=0;
        break;
      }
   }
```

Пример работы со стеком

```
• if(text[i]=='(')
•   Push_Stack(text[i],Top_Stack);
•  }
•  if(flag!=0 && Empty_Stack(Top_Stack))
•   printf("Верно!");
•  else printf("Неверно!");
•  Clear_Stack(Top_Stack);
•  printf("\n");
```

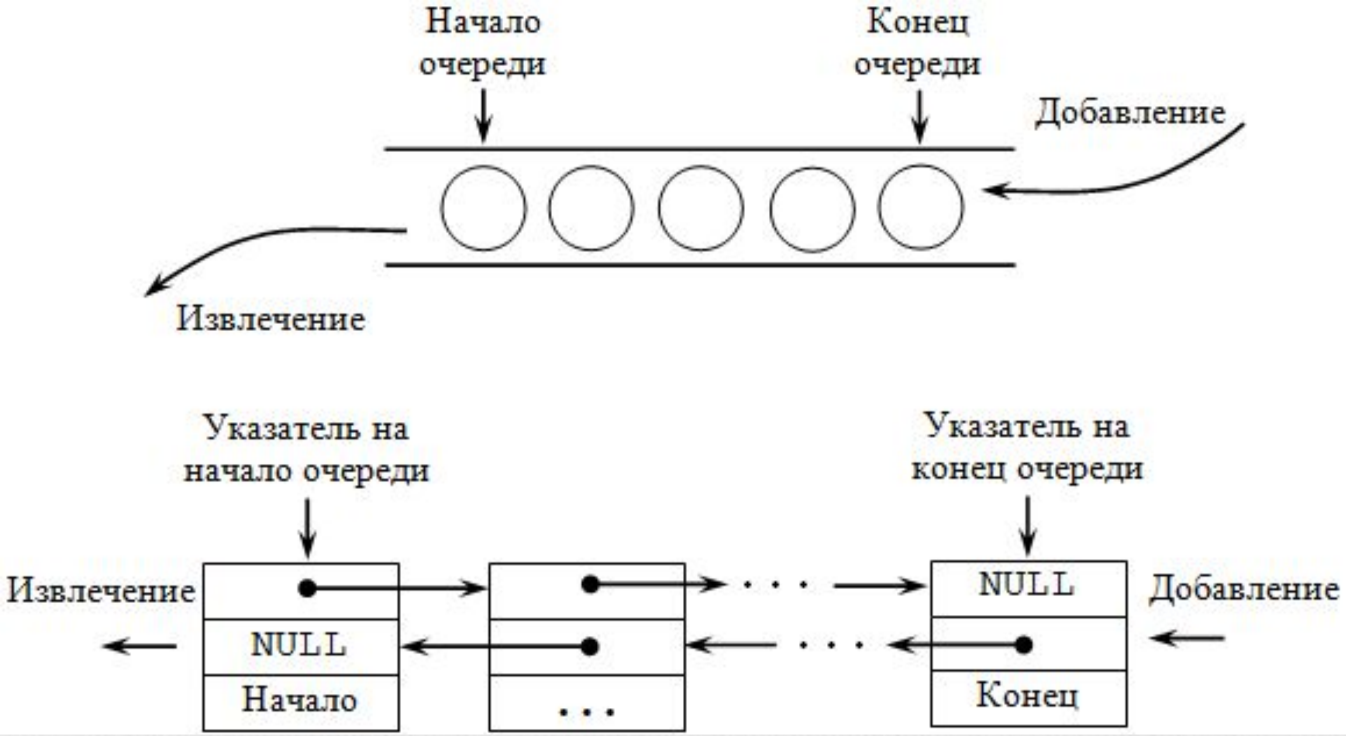
Очереди

- **Очередь – это структура данных, представляющая собой последовательность элементов, образованная в порядке их поступления. Каждый новый элемент размещается в конце очереди; элемент, стоящий в начале очереди, выбирается из нее первым. В очереди используется принцип доступа к элементам FIFO (First Input – First Output, "первый пришёл – первый вышел«).**

Очереди

- В очереди доступны два элемента (две позиции): начало очереди и конец очереди. Поместить элемент можно только в конец очереди, а взять элемент только из ее начала. Примером может служить обыкновенная очередь в магазине.

Очереди


Начало очереди
Конец очереди
Добавление
Извлечение
Указатель на начало очереди
Указатель на конец очереди
Извлечение
NULL
Начало
. . .
. . .
. . .
NULL
Конец
Добавление

Описание очереди

- Описание очереди выглядит следующим образом:
- struct имя_типа {
- информационное поле;
- адресное поле1;
- адресное поле2;
- };

Описание очереди

- где информационное поле – это поле любого, ранее объявленного или стандартного, типа;
- адресное поле1, адресное поле2 – это указатели на объекты того же типа, что и определяемая структура, в них записываются адреса первого и следующего элементов очереди.

Описание очереди

- Например:

- 1 способ: адресное поле ссылается на объявляемую структуру.
- struct list2 {
-         type pole1;
-         list2 *pole1, *pole2;
-       }

Описание очереди

- 2 способ: адресное поле ссылается на ранее объявленную структуру.
- struct list1 {
- type pole1;
- list1 *pole2;
- }
- struct ch3 {
- list1 *beg, *next ;

Организация очереди

Очередь как динамическую структуру данных легко организовать на основе линейного списка. Поскольку работа идет с обоими концами очереди, то предпочтительно будет использовать линейный двунаправленный список.

Организация очереди

- Хотя для работы с таким списком достаточно иметь один указатель на любой элемент списка, здесь целесообразно хранить два указателя – один на начало списка (откуда извлекаем элементы) и один на конец списка (куда добавляем элементы). Если очередь пуста, то списка не существует, и указатели принимают значение NULL.

Описание очереди

- Объявление очереди через объявление линейного двунаправленного списка:
- struct Queue {
-     Double_List *Begin;//начало очереди
-     Double_List *End; //конец очереди
-   };
- ……….
- Queue *My_Queue;//указатель на очередь

Основные операции, производимые с очередью

- Основные операции, производимые с очередью:

✓ создание очереди;

✓ печать (просмотр) очереди;

✓ добавление элемента в конец очереди;

✓ извлечение элемента из начала очереди;

✓ проверка пустоты очереди;

✓ очистка очереди.

Создание очереди

- Реализацию этих операций приведем в виде соответствующих функций, которые, в свою очередь, используют функции операций с линейным двунаправленным списком.
- //создание очереди
- void Make_Queue(int n, Queue* End_Queue){
- Make_Double_List(n,&(End_Queue->Begin),NULL);

Создание очереди

```
Double_List *ptr; //вспомогательный указатель
 ptr = End_Queue->Begin;
 while (ptr->Next != NULL)
   ptr = ptr->Next;
 End_Queue->End = ptr;
}
```

Печать очереди

```
//печать очереди
void Print_Queue(Queue* Begin_Queue){
  Print_Double_List(Begin_Queue->Begin);
}
```

Добавление элемента в конец очереди

- //добавление элемента в конец очереди
- void Add_Item_Queue(int NewElem, Queue* End_Queue){
- End_Queue->End = Insert_Item_Double_List(End_Queue->End,
- 0, NewElem)->Next;
- }

Извлечение элемента из начала очереди

- //извлечение элемента из начала очереди
- int Extract_Item_Queue(Queue* Begin_Queue){
- int NewElem = NULL;
- if (Begin_Queue->Begin != NULL) {
- NewElem = Begin_Queue->Begin->Data;

Извлечение элемента из начала очереди

- Begin_Queue->Begin=Delete_Item_Double_List(Begin_Queue->Begin,0);
- //удаляем вершину
- }
- return NewElem;
- }

Проверка пустоты очереди

- //проверка пустоты очереди
- bool Empty_Queue(Queue* Begin_Queue){
- return Empty_Double_List(Begin_Queue->Begin);
- }

Очистка очереди

- //очистка очереди
- void Clear_Queue(Queue* Begin_Queue){
- return Delete_Double_List(Begin_Queue->Begin);
- }

Пример работы с очередью

- Пример. Дана последовательность ненулевых целых чисел. Признаком конца последовательности является число 0. Найдите среди них первый наибольший отрицательный элемент. Если такого элемента нет, то выведите сообщение об этом.

Пример работы с очередью

- В данной задаче будем использовать основные операции для работы с очередью, рассмотренные ранее. Приведем главную функцию и функцию для реализации поиска первого наибольшего отрицательного элемента.

Пример работы с очередью

- //главная функция
- int _tmain(int argc, _TCHAR* argv[]){
- int n;
- Queue *My_Queue;
- My_Queue = new Queue();
- Make_Queue(1,My_Queue);

Пример работы с очередью

```
while (My_Queue->End->Data != 0){
    cout << "Введите значение ";
    cin >> n;
    Add_Item_Queue(n,My_Queue);
 }
```

Пример работы с очередью

- cout << "\nОчередь: \n";
-   Print_Queue(My_Queue);
-   Find_Max_Negative_Element(My_Queue);
-   system("pause");
-   return 0;
- }

Пример работы с очередью

- //функция поиска первого наибольшего отрицательного элемента
- void Find_Max_Negative_Element(Queue* Begin_Queue){
- int tmp;
- int max=Extract_Item_Queue(Begin_Queue);
-

Пример работы с очередью

- while (Begin_Queue->Begin->Data != 0) {
- tmp = Extract_Item_Queue(Begin_Queue);
- if (max > 0 || tmp < 0 && abs(tmp) < abs(max))
- max = tmp;
- }

Пример работы с очередью

- if (max > 0) printf("Элементов нет!");
- else printf("Есть такой элемент: %d", max);
- }